METODOLOGÍA DE LA PROGRAMACIÓN

Informática de Gestión

Profesor: Borja Fdez. Gauna
Despacho: Departamento de LSI
Emails: borja.fernandez@ehu.es
Apuntes y ejercicios: http://lsi.vc.ehu.es/bortx
Tfno.: 945 01 **3255**

ÍNDICE

0. INTRODUCCIÓN

El **desarrollo de software** consiste en, dado un **problema**, encontrar un **programa** (o conjunto de programas) que lo resuelva de forma correcta y eficientemente.

En general, el desarrollo de software tiene 4 fases:

1. Análisis: Dado un problema, inicialmente se hace el análisis del mismo, de manera que obtengamos una especificación clara y concisa de qué es lo que se quiere hacer.

2. Diseño: A partir de la especificación, se hace un diseño del programa que se va a hacer, después del cuál se obtendrán los algoritmos necesarios para poder escribir el programa.

3. Codificación: Una vez se tiene el diseño del programa, se procede a implementarlo en el/los lenguaje(s) de programación elegido(s).

4. Mantenimiento: Cuando el software ya está implantado en el lugar de implantación, se hace seguimiento del funcionamiento del programa, así como se toma nota de los posibles errores detectados para ser corregidos en una versión posterior.

Desgraciadamente, todos erramos y, debido a errores humanos en el proceso, a menudo se hace necesario volver a una fase anterior. Por ejemplo, una mala comprensión del problema, puede hacer que, habiendo empezado a implementar, se tenga que volver a la fase de análisis.

Cuanto más tarde se detecta un error, más caro (en términos de tiempo y dinero) resulta solucionarlo, por lo que se hace necesaria una **metodología de trabajo** para minimizar los errores humanos.

En la asignatura **Metodología de la Programación** vamos a ver diferentes conceptos y técnicas que van a facilitar el proceso de desarrollar software:

1. Especificación formal: nos permitirá definir de manera clara que no permita interpretaciones diferentes de qué es lo que se quiere hacer y qué requisitos se tendrán que cumplir para que el programa de los resultados deseados.

2. Verificación de programas: nos ayudará a comprobar que un programa cumple con su especificación, de manera que podamos detectar errores.

3. Documentación de programas: permitirá que el código pueda ser revisado y modificado por personas diferentes al autor sin perder excesivo tiempo en la comprensión del mismo.

4. Derivación formal de programas: sirve para escribir de manera razonada y casi

automática un programa a partir de las especificaciones que cumple.

5. Transformación de programas: nos permitirá transformar de manera automática programas ineficientes (recursivos) en otros que, cumpliendo las especificaciones originales, resulten más eficientes.

1. ASERCIONES LÓGICAS

1.1 Definiciones y notación

Estado (de cómputo) de un programa: Descripción completa de los valores asociados a todas las variables del programa en un momento determinado del cómputo. Por ejemplo, si un programa tiene en un momento de su ejecución las variables X e Y con los valores 5 y -24 respectivamente, su estado de cómputo en ese momento sería {X=5, Y=-24}.

Cómputo: Sucesión de estados de cómputo. Ejecución.

Aserciones: Son **afirmaciones** sobre el estado del programa, fórmulas o expresiones lógicas asociadas a un punto determinado de un programa, que nos sirven para expresar propiedades de las variables del programa.

Se suelen incluir en el programa entre las líneas de código como comentarios (/*…*/).

Ejemplo de aserciones asociadas al código de un programa (con un ejemplo posible de los estados de cómputo):

Código + Aserciones	Estado de cómputo
x= 5;	
/*x=5*/	{ x=5 }
y= x+2;	
/*x=5 ^ y=x+2*/	{ x=5, y=7 }
z= x+y;	
/*x=5 ^ y=x+2 ^ z=x+y*/	{ x=5, y=7, z=12 }

Mientras un estado de cómputo refleja un único estado posible de las variables (un conjunto de valores), una aserción es más genérica y puede ser cierta para diferentes estados de cómputo.

Por ejemplo, la aserción **/* x=y*2 */** puede ser cierta tanto para el estado de cómputo **{x=12,y=6}** como para **{x=26,y=13}**. En este caso, la aserción representa un número casi-infinito de estados de cómputo posibles (no es infinito porque en una máquina los números tienen una resolución finita).

Cuando no se imponga ninguna *precondición* (ver el Tema 2) o no se conozca ninguna propiedad de las variables (por ejemplo al inicio de un programa), la aserción asociada será

/*true*/, ya que se cumplirá (el valor de verdad asociado será verdadero) para cualquier estado de cómputo posible.

1.2 Lógica de Primer Orden (LPO)

A la hora de escribir aserciones, necesitamos un lenguaje (**Lógica de Primer Orden**) del que valernos para expresar las propiedades que cumplen las variables, con su correspondiente alfabeto, sintaxis y semántica.

a) Alfabeto.

- **Variables**: contienen valores modificables durante el cómputo.
  P.ej.: x, a, A[], …
- **Constantes**: representan valores concretos y fijos.
  P.ej.: 1, 10, 'a', PI, …
- **Funciones**: operan sobre valores devolviendo otro valor.
  P.ej.: +, -, %, /, Σ, Π, …
- **Predicados**: funciones que devuelven un valor **booleano** (true o false)
  P.ej.: <, =, >, par, impar, …

b) Sintaxis.

- **Términos**: elementos atómicos que combinados forman una fórmula.
  P.ej.: x puede ser una variable
  c puede ser una constante
  $f(t_1,...,t_n)$ es una llamada a la función f n-aria y $t_1 \ldots t_n$ términos
- **Fórmulas**: conjunción de términos que se evalúa como true o false.
  átomos: p(t1,…,tn) con p predicado n-ario y n términos
  compuestas: siendo ψ, Φ fórmulas, por ejemplo:
  conectivos: ¬ ψ, ψ^Φ, ψ→Φ,…
  cuantificadores: ∀x, ∃x

c) Semántica. Asocia a cada fórmula un valor booleano dependiendo del estado de cómputo de las variables.

Diremos que una **fórmula está definida** o tiene valor si todas sus variables libres (variables del programa, no ligadas a un cuantificador) tienen un valor asociado en dicho estado.

1.2.1 Conectivos

Unen fórmulas formando fórmulas compuestas. En la tabla siguiente están todos los conectivos; así como sus correspondientes valores de verdad, que utilizaremos este curso.

φ	ψ	¬φ (not)	φ∨ψ (or)	φ∧ψ (and)	φ→ψ (si)	φ↔ψ (sii)
T	**T**	F	T	T	T	T
T	**F**	F	T	F	F	F
F	**T**	T	T	F	T	F
F	**F**	T	F	F	T	T

1.2.2 Cuantificadores

Abreviatura para una serie. Utilizaremos ∀ (para todo), ∃ (existe), Π(productorio), Σ (sumatorio) y N (cardinal o recuento).

Todos los cuantificadores tendrán una **variable ligada**, que no tendrá significado fuera del ámbito del cuantificador. Servirá para expresar el alcance de una propiedad y tendrá un **dominio**: un conjunto de valores que se le darán a la variable ligada a los que se referirá la propiedad.

1.2.2.1 Σ (sumatorio)

Lo denotaremos $\sum_{D(x)} f(x)$ donde D(x) es el dominio o conjunto de valores que tomará x (o cualquiera que sea la variable utilizada para definir el dominio), siendo ésta una **variable ligada** al sumatorio, que no existirá fuera del sumatorio.

El sumatorio tiene **asociada la suma** (+). Si el dominio es vacío (dicho de otra manera, si no se le da valores a i), devolverá **0, elemento neutro de la suma**.

Por ejemplo,

$$\sum_{i=1}^{3} i^2 = 1^2 + 2^2 + 3^2 = 14$$

$$\sum_{i=1}^{n} A(i) = A(1) + A(2) + A(3) + ... + A(n)$$

$$\sum_{j=5}^{3} 2j^3 + j^2 + 3j = 0$$

En este último caso, el dominio es vacío por no haber ninguna j que cumpla 5<=j<=3.

1.2.2.2 Π (productorio):

Similarmente al sumatorio, lo denotaremos $\prod_{D(x)} f(x)$ donde D(x) será el dominio o conjunto de valores que tomará x (o cualquiera que sea la variable utilizada para definir el dominio), siendo ésta una **variable ligada** al productorio.

Tiene **asociado el producto** (*). Si el dominio es vacío (dicho de otra manera, si no se le da valores a i), devolverá **1, elemento neutro del producto**.

Por ejemplo,

$$\prod_{i=2}^{5} i = 2*3*4*5 = 120$$
$$\prod_{k=i}^{j} A(k)^2 = A(i)^2 * A(i+1)^2 * A(i+2)^2 * ... * A(j)^2$$
$$\prod_{i=1}^{0} i^2 + i = 1 \quad \text{(dominio vacío)}$$

1.2.2.3 Cuantificador universal: ∀ (para todo)

Será de la forma $\forall x(D(x) \rightarrow P(x))$, donde *x* es la variable ligada, D(x) es el dominio de la variable ligada al cuantificador y P(x) la propiedad.

Si la propiedad P(x) se cumple para todo el dominio de la variable ligada, el cuantificador se evaluará como true, y en caso contrario como false.

Tiene **asociada la operación de implicación** (→), que se utilizará para unir dominio y propiedades . En caso de dominio vacío, se evaluará como **true**.

Por ejemplo,

$\forall x(10 < x < 20 \rightarrow x^2 > 0) = true$ (todos los números enteros entre el 10 y el 20 elevados al cuadrado dan un número positivo)
$\forall x(x > 0 \rightarrow x \; mod \; 2 = 0) = false$ (no todos los números son múltiplos de 2)
$\forall i(15 < i < 10 \rightarrow A(i) = 0) = true$ (dominio vacío)

1.2.2.4 Cuantificador existencial: ∃ (existe)

Tendrá la forma $\exists x(D(x) \wedge P(x))$, donde x es la variable ligada, D(x) es el dominio de la variable ligada al cuantificador y P(x) la propiedad.

Si la propiedad P(x) se cumple para alguno de los valores del dominio de la variable ligada, el cuantificador se evaluará como true, y en caso contrario como false.

Tiene **asociada la operación lógica and** ($\wedge$). Si el dominio fuese vacío, se evaluaría como **false**.

Por ejemplo,

$\exists x(-3<x<2 \wedge x>0)=true$ (el 1 está en el dominio y es positivo)
$\exists x(1 \le i \le n \wedge A[i]>0)$ (será cierto si el vector tiene algún número positivo)
$\exists i(1<i \le 1 \wedge i>x)=false$ (dominio vacío)

1.2.2.5 Cuantificador de recuento: $\aleph$ (cardinal)

Se expresará $\aleph x(D(x) \rightarrow P(x))$, siendo D(x) el dominio de la variable ligada *x* y P(x) la propiedad.

A diferencia de los dos cuantificadores anteriores, éste no se evalúa como un valor booleano, sino como un número natural: el número de elementos del dominio que cumplen la propiedad P(x).

En caso de ser el dominio vacío, se evaluará como 0, ya que no habrá elementos que contar.

Por ejemplo,

$\aleph i(1 \le i \le 8 \rightarrow i \, mod \, 3=0)=2$ (hay 2 múltiplos de 3 entre el 1 y el 8)
$\aleph i(1 \le i \le n \rightarrow A[i] \, mod \, 2=0)$ (devolverá el número de elementos pares del vector A[1..n])
$\aleph j(2<j \le 2 \rightarrow j^2=20)=0$ (dominio vacío)

1.2.3 Predicados

A la hora de intentar representar en LPO enunciados de gran complejidad o longitud, es a veces deseable afrontar el problema dividiéndolo en conceptos más pequeños.

De similar manera a como definimos funciones cuando programamos para hacer más legible y reutilizable el código, podemos definir predicados que nos ayuden a hacer más fácil de entender la fórmula y poder, además, reutilizarlos en diferentes ejercicios.

Siguiendo la analogía con las funciones a la hora de programar, definiremos un predicado $p(t_{1,\dots}, t_n)$ de igual manera que le damos un nombre a una función y definimos cuáles serán sus parámetros $(t_{1,\dots}, t_n)$.

Aunque generalmente usaremos predicados por la naturaleza booleana de las aserciones, puede que nos interese definir una función en vez de un predicado. Recordemos que, en lo que a esta asignatura atañe, la diferencia entre ambos es que, mientras un predicado devuelve un valor booleano, una función devolverá un valor numérico.

Ejemplos:

- Expresar en LPO que todos los elementos de A[1..n] son primos:

Primero podemos definir el predicado esPrimo(x) que nos diga si un número cualquiera (x) es primo:

$$esPrimo(x) \equiv \forall i\,(1 < i < x \rightarrow x \bmod i \neq 0)$$

Una vez definido resulta más fácil generalizar para todo el vector:

$$\forall\ j\,(1 \leq j \leq n \rightarrow esPrimo(A[j]))$$

- Expresar en LPO que el vector A[1..n] está ordenado de menor a mayor:

Si está ordenado de menor a mayor, podemos definir el predicado esMáximo(x,A,i,j) que nos dirá si x es el máximo de A[i..j] para luego generalizarlo al vector entero.

$$esMaximo(x, A, i, j) \equiv \forall k\,(i \leq k \leq j \rightarrow A[k] \leq x)$$

Ahora generalizamos a todo el vector:

$$\forall i\,(1 < i \leq n \rightarrow esMaximo(A[i], A, 1, i-1))$$

NOTA: Siempre que utilicemos un predicado o una función, habrá que definirlo/a previamente, igual que una función tiene que haberse definido antes de llamarla cuando estemos programando.

1.2.4 Operación de sustitución

Dada la fórmula Φ , la variable x y el término t, definiremos Φ_x^t como la fórmula resultante de sustituir todas las apariciones libres de x por t en Φ . Como aparición libre entendemos todas aquellas que no sean variables ligadas del mismo nombre.

Si el término t tuviese alguna variable que apareciese como ligada en ϕ, habrá que renombrarlas antes de hacer la sustitución.

Esta es una operación que utilizaremos ampliamente cuando veamos en el cálculo de Hoare el Axioma de Asignación (AA).

Ejemplos:

$$(x^2+2x+17)_x^y \equiv y^2+2y+17$$
$$(x<5 \wedge 1<y<5)_x^{x-5} \equiv x-5<5 \wedge 1<y<5 \equiv x<10 \wedge 1<y<5$$
$$(\sum_{k=i}^{j} A[k])_i^{i+1} \equiv \sum_{k=i+1}^{j} A[k]$$

1.2.5 Fórmulas débiles y fuertes

Diremos que la fórmula φ es más fuerte que ψ (o que ψ es más débil que φ) si y sólo si el conjunto de valores que hacen cierto φ es un subconjunto de los que hacen cierto ψ:

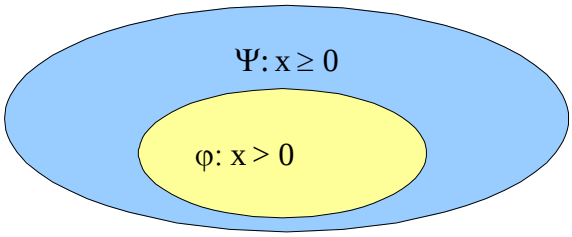
φ **más fuerte que** $\psi \leftrightarrow$ **{s | s (**φ**)= true}** $\subseteq$ **{s | s (**ψ**)=true}**

Dicho más intuitivamente, la fórmula más fuerte es aquella que tiene un menor número de estados que hacen cierta la fórmula.

Por ejemplo,

x>0 es más fuerte que x>=0

- ya que la segunda, además de todos los valores que hacen cierto x>0, es cierto para x=0.
- $x \geq 0 \rightarrow x > 0 \vee x = 0$



Ejercicios LPO I

Expresa los siguientes enunciados en Lógica de Primer Orden:

1. *x* es positivo.
2. *x* es nulo.
3. *x* es natural.
4. *x* es múltiplo de *y*. Por ejemplo, 10 es múltiplo de 5.
5. *x* es divisor de *y*. Por ejemplo, 5 es divisor de 10.
6. *x* es múltiplo de 5, pero no lo es de 10.
7. *x* es potencia de 2.
8. *x* es un número primo.
9. *z* es la suma de los n primeros números naturales.
10. *z* es la suma de una secuencia de números naturales que comienza por 1.
11. En el vector *A[1..n]* no hay ningún elemento negativo.
12. La suma de todos los elementos de la sección *A[i..j]* es mayor que cada uno de los elementos del vector. *Nota: los elementos pueden ser negativos.*
13. El vector *B[1..n]* está ordenado de manera decreciente.
14. Hay un único elemento en *A[1..n]* igual a su índice.
15. *max* es el máximo de la sección *A[i..j]*.
16. En el vector *B[1..n]* no se repite ningún elemento.
17. El vector *A[1..n]* es capicúa.
18. La suma de los valores absolutos de todos los elementos de *A[1..n]* es 0.
19. x aparece en el vector *A[1..n]* por primera vez en la posición i.
20. El vector *A[1..n]* contiene los n primeros números de la serie de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21, ...)
21. En el vector *A[1..n]* tenemos representados los dígitos del número natural *x*.
22. En los vectores *A[1..n]* y *B[1..m]* tenemos los dígitos de dos números naturales que sumados dan *x*.

Ejercicios LPO II

Decide si las siguientes implicaciones son correctas o no:

1. $x<7 \rightarrow x<9$
2. $x<7 \rightarrow x \leq 6$
3. $x+1<y \rightarrow x<y+1$
4. $j-i \geq 0 \rightarrow j-i+1-1 \geq 0$
5. $i<6 \wedge i>2 \rightarrow i<5$
6. $x<7 \rightarrow x+1 \leq 7$
7. $1<i<n \rightarrow 1 \leq i \leq n$
8. $1 \leq i \leq n \rightarrow 1<i+1<n$
9. $\text{sum}=\sum_{i=1}^{n-1} A[i] \rightarrow (\text{sum}=\sum_{i=1}^{n} A[i])-A[n]$
10. $\text{sum}=\sum_{i=1}^{n-1} A[i] \rightarrow \text{sum}+A[n]=\sum_{i=1}^{n} A[i]$

Soluciones propuestas ejercicios LPO I

1. $positivo(x) \equiv x > 0$
2. $nulo(x) \equiv x = 0$
3. $natural(x) \equiv x \geq 0 \equiv x > 0 \vee x = 0 \equiv positivo(x) \vee nulo(x)$
4. $multiplo(x, y) \equiv x \; mod \; y = 0 \equiv \exists z (z > 1 \wedge x = y * z)$
5. $divisor(x, y) \equiv y \; mod \; x = 0 \equiv \exists z (z > 1 \wedge y = x * z) \equiv multiplo(y, x)$
6. $imparM5(x) \equiv x \; mod \; 5 = 0 \wedge x \; mod \; 10 \neq 0 \equiv multiplo(x, 5) \wedge \neg multiplo(x, 10)$
7. $pot2(x) \equiv \exists y (y \geq 1 \wedge x = 2^y)$
8. $primo(x) \equiv x > 1 \wedge \forall i (1 < i < x \rightarrow x \; mod \; i \neq 0)$
9. $esSuma(z, n) \equiv z = \sum_{i=1}^{n} i$
10. $esUnSuma(z) \equiv \exists n (n \geq 1 \wedge z = \sum_{i=1}^{n} i)$
11. $ningunNegativo(A, n) \equiv \forall i (1 \leq i \leq n \rightarrow A[i] \geq 0) \equiv \neg \exists i (1 \leq i \leq n \wedge A[i] < 0)$
12. $sumaSección(A, i, j) \equiv \sum_{k=i}^{j} A[k]$

    $suSeMaCaEl(A, i, j) \equiv \forall k (1 \leq k \leq n \rightarrow sumaSección(A, 1, n) > A[k])$
13. $decreciente(B, n) \equiv \forall i (1 \leq i < n \rightarrow B[i] \geq B[i+1]) \equiv \neg \exists i (1 \leq i < n \wedge B[i] < B[i+1])$
14. $unoIgualIndice(A, n) \equiv \aleph i (1 \leq i \leq n \rightarrow A[i] = i) = 1$
15. $maxEs(max, A, i, j) \equiv \exists k (i \leq k \leq j \wedge A[k] = max) \wedge \forall k (i \leq k \leq j \rightarrow A[k] \leq max)$
16. $apariciones(x, A, n) \equiv \aleph k (1 \leq k \leq n \rightarrow A[k] = x)$

    $distintos(A, n) \equiv \forall i (1 \leq i \leq n \rightarrow apariciones(A[i], A, n) = 1)$
17. $capicúa(A, n) \equiv \forall i (1 \leq i \leq n/2 \rightarrow A[i] = A[n - i + 1])$
18. $sumaValAbsNula(A, n) \equiv \sum_{i=1}^{n} |A[i]| = 0 \equiv \forall i (1 \leq i \leq n \rightarrow A[i] = 0)$
19. $primeraI(x, i, A, n) \equiv \forall k (1 \leq k < i \rightarrow A[k] != x) \wedge A[i] = x \wedge 1 \leq i \leq n$
20. $fiboA(A, n) \equiv A[1] = 1 \wedge A[2] = 1 \wedge \forall i (2 < i \leq n \rightarrow A[i] = A[i-1] + A[i-2])$
21. $dig(A, n) \equiv \forall k (1 \leq k \leq n \rightarrow 0 \leq A[k] < 10)$

    $numDig(A, n) \equiv \sum_{i=1}^{n} A[i] * 10^{n-i}$

    $vectorDe(A, n, x) \equiv dig(A, n) \wedge x = numDig(A, n)$

22. $suma(A, n, B, m, x) \equiv dig(A, n) \wedge dig(B, m) \wedge x = nDig(A, n) + nDig(B, m)$

Soluciones propuestas ejercicios LPO II

1. $x<7 \rightarrow x<9$ Cierto
2. $x<7 \rightarrow x \leq 6$ Cierto
3. $x+1<y \rightarrow x<y+1$ Cierto
4. $j-i \geq 0 \rightarrow j-i+1-1 \geq 0$ Cierto
5. $i<6 \wedge i>2 \rightarrow i<5$ Falso
6. $x<7 \rightarrow x+1 \leq 7$ Cierto
7. $1<i<n \rightarrow 1 \leq i \leq n$ Cierto
8. $1 \leq i \leq n \rightarrow 1<i+1<n$ Falso
9. $\text{sum}=\sum_{i=1}^{n-1} A[i] \rightarrow (\text{sum}=\sum_{i=1}^{n} A[i])-A[i]$ Falso
10. $\text{sum}=\sum_{i=1}^{n-1} A[i] \rightarrow \text{sum}+A[n]=\sum_{i=1}^{n} A[i]$ Cierto

2. ESPECIFICACIÓN

Una de los principales motivos para volver a una fase anterior en el ciclo de desarrollo de software (con el consiguiente coste tiempo-dinero) suele ser una incorrecta comprensión del problema que se quiere solucionar.

Esto generalmente está causado por la imprecisión y falta de rigurosidad a la hora de especificar el objetivo del programa.

Para evitar, en la medida de lo posible, este tipo de problemas, utilizaremos la especificación formal: así definiremos de forma clara y concisa qué se tiene que cumplir antes de llamar a un programa y qué es lo que devolverá el programa (o función).

2.1 Especificación Pre-Post

Llamaremos especificación Pre-Post a aquella que defina cuál es la **precondición** y cuál es la **postcondición** de un trozo de código, definiendo como precondición la fórmula que define qué se tiene que cumplir antes de ejecutar el código para que al finalizar se cumpla la postcondición, que es la fórmula que describe la relación entre los datos de entrada y los resultados: lo que se cumplirá al finalizar el programa.

En términos más sencillos, una especificación Pre-Post determina qué tiene que garantizar quien llame al código antes de llamarlo (precondición) para que el programa pueda cumplir su cometido (postcondición).

En general, escribiremos **/* φ */ P /* ψ */** para indicar que el programa P tiene la precondición φ y la postcondición ψ.

Por ejemplo:

$$/*\, 1 \le i < n \wedge suma = \sum_{j=1}^{i} A[j] \,*/ \equiv \Phi$$

```
i = i + 1;
suma = suma + A[i];
```

$$/*\, 1 < i \le n \wedge suma = \sum_{j=1}^{i} A[j] \,*/ \equiv \Psi$$

En este ejemplo, siempre que al principio del programa se cumpla que *i* esté dentro del rango del array *A* (sin ser la última posición) y que tengamos en la variable *suma* la suma de los primeros *i* elementos (precondición: ϕ), después de ejecutarse el código siempre se cumplirá que *i* seguirá estando dentro del rango del array y que en la variable *suma* tendremos la suma de los primeros i elementos (postcondición: ψ) , habiendo sumado +1 a i.

Nos interesará además que la **precondición sea lo más débil posible** (lo menos exigente posible) y que la **postcondición sea lo más fuerte posible** (que describa la relación entre datos de entrada y resultado de la manera más exacta y concisa posible).

Así, en el ejemplo anterior, $/*1 \le i < n \wedge suma = \sum_{j=1}^{i} A[\,j\,] \wedge \mathbf{suma} > \mathbf{0} */$ sería una **precondición demasiado fuerte**, ya que no es necesario que suma sea mayor que 0 para que después de las 2 instrucciones se cumpla la postcondición.

De igual manera, $/*1 < i \le n */$ sería una **postcondición demasiado débil**, ya que no expresa la relación entre la variable suma y los elementos del vector.

A modo de resumen, la precondición debe establecer las condiciones mínimas que se deben cumplir para que al final del programa se llegue a la postcondición, que deberá expresar todas las propiedades conocidas entre los datos de entrada y los resultados.

3. VERIFICACIÓN

La verificación es la técnica que vamos a ver más en profundidad y la que ocupará la mayoría del tiempo de clase. Nos permitirá demostrar formalmente (verificar) que un programa es correcto.

La corrección es un término subjetivo y ambiguo en el mundo real, pero en cuanto a esta asignatura atañe, diremos que un programa es correcto si respeta su especificación formal.

3.1 Verificación de corrección parcial y total

Diremos que un programa es **parcialmente correcto** si, siempre que al inicio del programa se cumpla la precondición, en caso de que el programa acabe, siempre acabará satisfaciendo la postcondición.

Escribiremos $/*\Phi*/\ P\ /*\Psi*/$ para decir que el programa P es parcialmente correcto respecto a la especificación formada por ϕ (precondición) y ψ (postcondición).

Diremos que un programa es **totalmente correcto** si, siempre que al inicio de la ejecución se respete la precondición, el programa acabará satisfaciendo la postcondición. Respecto a la corrección parcial se añade además la garantía de que el programa siempre acabará.

Denotaremos $/*\Phi*/[\,P\,]/*\Psi*/$ la corrección total del programa P respecto a su especificación Pre-post.

Si definimos $Term(\Phi, P)$ como el predicado que nos dice si toda ejecución de P que cumpla inicialmente la precondición ϕ va a acabar en un número finito de pasos, podemos afirmar que $/*\Phi*/[\,P\,]/*\Psi*/ \equiv /*\Phi*/\ P\ /*\Psi*/ + Term(\Phi, P)$.

3.2 Cálculo de Hoare

Para verificar la corrección total o parcial de un programa utilizaremos el cálculo de Hoare, que utilizará **axiomas** (por ejemplo, el Axioma de Asignación) y **reglas de inferencia** (RCN, RCP etc...).

A la hora de verificar la corrección de un programa, el proceso a seguir es, a partir de la precondición, ir deduciendo, mediante axiomas y reglas de inferencia, las propiedades entre variables que se mantendrán en los diferentes puntos del programa. El objetivo final siempre será demostrar que al final del programa se cumplirá la postcondición.

Los **axiomas** serán propiedades básicas indemostrables de la forma $/*\Phi*/\ P\ /*\Psi*/$, donde P podría ser, por ejemplo, una asignación. Si en algún paso anterior se hubiese demostrado que antes de P se cumplía ϕ , quedará demostrado que después de la asignación se cumplirá ψ.

Las **reglas de inferencia** tendrán la forma $\dfrac{/*...*/\ \boldsymbol{P_1}\ /*...*/\ ,/*...*/\ \boldsymbol{P_2}\ /*...*/,...}{/*...*/\ \boldsymbol{P_1}\times\boldsymbol{P_2}\ /*...*/}$, y, habiendo sido previamente demostradas las propiedades $/*...*/\ \boldsymbol{P_1}\ /*...*/$, $/*...*/\ \boldsymbol{P_2}\ /*...*/$, ... la regla de inferencia demostrará que se verificará $/*...*/\ \boldsymbol{P_1}\times\boldsymbol{P_2}\ /*...*/$.

3.3 Axioma de Asignación (AA)

El primer axioma que estudiaremos será el de asignación. Este axioma nos permitirá inferir las propiedades que se cumplirán después de una asignación.

En una **asignación** distinguiremos dos partes: la **variable x** donde se almacenará el valor (parte izquierda de la asignación) y el **término t** de la parte derecha que se evaluará como un valor (generalmente numérico).

$$\boldsymbol{x = t\ ;}$$

Antes de realizar una asignación conoceremos una serie de propiedades sobre las variables que forman parte del *término t*. Todo aquellas propiedades del *término t* que se conozcan antes de la asignación, las cumplirá la *variable x* después de la asignación.

Así, en el siguiente ejemplo:

$$\begin{array}{c} \Phi = /^*\ z\ mod\ 2 = 0\ ^*/ \\ \boldsymbol{x = z\ ;} \\ \Psi = /^*\ x\ mod\ 2 = 0 \wedge z\ mod\ 2 = 0\ ^*/ \end{array}$$

Sabiendo que z es un número par antes de la asignación, después de ella siempre se cumplirá que **x** será un número par. Además, seguiremos sabiendo lo mismo de *z* que antes de la asignación, puesto que ni la *variable z* ni ninguna variable libre de ϕ ha sido modificada en la asignación.

En este otro ejemplo, en cambio:

$$\begin{array}{c} \Phi \equiv /^*\ x - 2 > 0\ ^*/ \\ \boldsymbol{x = x - 2\ ;} \\ \Psi \equiv /^*\ x > 0 \wedge \cancel{x - 2 > 0}\ ^*/ \end{array}$$

Después de la asignación, no podemos afirmar que *x-2>0* se siga cumpliendo, ya que la *variable libre x* (que aparece en el *término t*) ha cambiado por ser además la variable donde se guardará el resultado.

Enunciaremos el **Axioma de Asignación** de la siguiente manera:

$$/^*\ \mathbf{def(t)} \wedge \Phi_x^t \wedge \Omega\ ^*/\ x = t\ ;\ /^*\ \Phi \wedge \Omega\ ^*/\quad (\mathbf{AA})$$

donde:

- **def(t)** es una fórmula que afirma que la evaluación del *término t* no produce error y está definido. ***Si contuviese a la variable x, no se podría afirmar después de la asignación*** y

puede no estar presente (o lo que es lo mismo, ser **true**).

- **Ω** será una fórmula donde **no aparecerá la *variable libre x* pero sí podrá aparecer *el término t***. Al ser independiente de la *variable x*, no cambiará después de la asignación.

- Φ_x^t será aquella fórmula que expresará las propiedades del término t antes de la asignación. En ella **aparecerán todas las variables libres del *término t* y podrá aparecer la *variable x***. Después de la asignación se cumplirán las mismas propiedades que se cumplían antes de la asignación para el *término t* para la *variable x* (Φ).

A continuación veremos unos cuantos ejemplos donde aplicaremos el AA para los distintos casos:

Ejemplo 1:

```
/* y≠0 ∧ z/y=2 ∧ v=12 */
z=z/y;
/* y≠0 ∧ z=2 ∧ v=12 */
```

En este ejemplo, el *término t* será la expresión *z/y* y la variable será *z*, que también aparecerá en el *término t.*

$def(t) \equiv y \neq 0$ ya que nos asegura que la división no nos dará un resultado indefinido y, como la *variable y* no ha cambiado en la asignación, se seguirá cumpliendo después.

Igualmente, $\Phi_x^t \equiv z/y = 2$ y $\Phi \equiv z = 2$. Describe la única propiedad conocida del *término z/y* antes de la asignación, que después pasará a ser la propiedad que cumplirá la *variable z*.

Finalmente, $\Omega \equiv v = 12$, ya que la fórmula no contiene la *variable libre z* ni el *término t.*

Ejemplo 2:

```
/* y≠0 ∧ y/2=16 */
z=y/2;
/* y≠0 ∧ z=16 ∧ y/2=16 */
```

En este ejemplo no existe def(t), ya que la división entre 2 nunca dará error..

$\Phi_x^t \equiv y/2 = 16$ y $\Phi \equiv z = 16$. En este caso, como la *variable z* no aparece en $\Phi_x^t \equiv y/2 = 16$, se cumplirá también que $\Omega \equiv y/2 = 16$ y se seguirá cumpliendo después de la asignación.

3.4 Deducciones lógicas

Desgraciadamente, el Axioma de Asignación no es suficiente para inferir propiedades después de una asignación, ya que Φ_x^t no siempre estará expresada tan claramente en función del término t como en los ejemplos anteriores.

En el siguiente ejemplo:

```
/* x = y/2 */
z = 2x ;
/* ??? */
```

No podemos inferir ninguna propiedad ya que en la aserción incial no aparece el término *2x*.

Sin embargo, utilizando deducciones lógicas, podríamos resolver parte de este tipo de problemas, ya que si sabemos que *x=y/2*, podremos deducir que *2*x=y*, con la que ya podríamos aplicar el Axioma de Asignación.

Veamos pues algunas de las **deducciones lógicas** (transformaciones que no cambian el *valor de verdad* de la fórmula original) que utilizaremos en adelante:

- $A \wedge B \rightarrow A$ (Simplificación)
- $A \rightarrow A \vee B$ (Adición)
- $((A \rightarrow B) \wedge A) \rightarrow B$ (Modus Ponens)
- $((A \rightarrow B) \wedge \neg B) \rightarrow \neg A$ (Modus Tollens)
- $((A \vee B) \wedge \neg B) \rightarrow A$ (Silogismo disyuntivo)
- $((A \rightarrow B) \wedge (B \rightarrow \Omega)) \rightarrow (A \rightarrow \Omega)$ (Transitividad)
- $((A \rightarrow B) \wedge B) \rightarrow A$ (Falacia: INCORRECTA)

Adicionalmente, a menudo utilizaremos **equivalencias tautológicas** (o **lógicas**) como las siguientes:

- $\neg(\neg A) \rightarrow A$
- $(A \rightarrow B) \Leftrightarrow (\neg A \vee B)$

- $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$
- $(A \wedge B) \vee (A \wedge \neg B) \Leftrightarrow A$
- $x < 5 \rightarrow x - 5 < 0$
- $(1 \leq i+1 \leq n) \rightarrow (0 \leq i < n) \rightarrow (1 \leq i < n+1) \rightarrow (1 \leq i \wedge i < n+1)$

Ahora que ya conocemos cuáles serán las herramientas que utilizaremos para transformar las fórmulas volveremos al ejemplo del principio:

```
/* x = y/2 */
z = 2x ;
/* ??? */
```

Como decíamos, no podemos aplicar directamente el Axioma de Asignación, ya que la aserción anterior a la asignación no está en función de *2x*. Pero ahora que hemos visto cómo transformar las fórmulas podemos decir:

1. $x = y/2 \rightarrow 2x = y$

Y ahora sí podemos aplicar AA:

2. /* 2x = y */ **z = 2x ;** /* z = y */ (***AA***)

Para acabar verificando la corrección del ejemplo sólo nos hará falta una regla de inferencia que nos permita unir ambos pasos: la **Regla de la Consecuencia** (RCN).

3.5 Regla de la Consecuencia (RCN)

Esta regla nos permitirá unir deducciones lógicas con las aserciones inferidas (resultados de las reglas de inferencia).

$$\frac{\Phi \rightarrow \Phi_1,\ /*\,\Phi_1\,*/\ \mathrm{P}\ /*\,\Psi_1\,*/,\ \Psi_1 \rightarrow \Psi}{/*\,\Phi\,*/\ \mathrm{P}\ /*\,\Psi\,/*}\ (\boldsymbol{RCN})$$

Dependiendo de si necesitamos hacer deducciones sólo antes o después de aplicar la regla de inferencia correspondiente, podremos utilizar las siguientes dos variantes:

a) $$\frac{\Phi \rightarrow \Phi_1,\ /*\,\Phi_1\,*/\ \mathrm{P}\ /*\,\Psi_1\,*/}{/*\,\Phi\,*/\ \mathrm{P}\ /*\,\Psi_1/*}\ (\boldsymbol{RCN})$$

b) $$\frac{/*\,\Phi\,*/\ \mathrm{P}\ /*\,\Psi_1\,*/,\ \Psi_1 \rightarrow \Psi}{/*\,\Phi\,*/\ \mathrm{P}\ /*\,\Psi\,/*}\ (\boldsymbol{RCN})$$

Gracias a la **RCN** podemos transformar aserciones y lo que verifiquemos para las aserciones transformadas quedará verificado para las originales, por lo que ya podemos retomar el ejemplo con el que empezamos y acabar verificando su corrección:

1. $x = y/2 \rightarrow 2x = y$
2. $/*2x = y\,*/\ \boldsymbol{z = 2x;}\ /*z = y*/\ (\boldsymbol{AA})$
3. $/*x = y/2\,*/\ \boldsymbol{z = 2x;}\ /*z = y*/\ (\boldsymbol{1,2\,RCN})$

De esta manera, en el tercer paso hemos verificado que, siempre que incialmente se cumpla *x=y/2*, después de la asignación siempre se verificará *z=y*.

Generalmente, cuando tengamos una asignación, **pondremos la aserción anterior a ella en función de la parte derecha de la asignación** (*término t* del AA) para poder aplicar **AA**, y luego uniremos ambos pasos con la **RCN**.

Ejemplo completo de cálculo de Hoare:

$$/*\,suma = \sum_{j=1}^{i} A[\,j\,] \wedge i < n\,*/$$
$$\boldsymbol{i = i + 1;}$$
$$/*\,suma = \sum_{j=1}^{i-1} A[\,j\,] \wedge i \leq n\,*/$$

Primero ponemos la aserción inicial en función de *i+1* (*término t*):

1. $suma = \sum_{j=1}^{i} A[j] \wedge i < n \rightarrow suma = (\sum_{j=1}^{i+1-1} A[j]) \wedge i+1 \leq n$

Ahora podemos aplicar AA sustituyendo *i+1* (*término t*) por *i* (*variable x*):

2. $/* suma = (\sum_{j=1}^{\boldsymbol{i+1-1}} A[j]) \wedge \boldsymbol{i+1} \leq n */ \ \boldsymbol{i = i+1;} \ /* suma = (\sum_{j=1}^{\boldsymbol{i-1}} A[j] \wedge \boldsymbol{i} \leq n) */ \ (\boldsymbol{AA})$

Finalmente, unimos los dos pasos con la RCN indicando qué propiedades hemos utilizado para aplicarla (en este caso las propiedades demostradas en los pasos 1 y 2):

3. $/* suma = \sum_{j=1}^{i} A[j] \wedge i < n */ \ \boldsymbol{i = i+1;} \ /* suma = \sum_{j=1}^{i-1} A[j] \wedge i \leq n */ (\mathbf{1,2}\ \boldsymbol{RCN})$

Ejercicios Verificación (I)

Verifica los siguientes programas:

1. $$/* y=5 */$$
   $$\mathbf{x = y * y + y;}$$
   $$/* x=30 */$$

2. $$/* x=y+5 */$$
   $$\mathbf{z = x + 5;}$$
   $$/* x=y+5 \wedge z=y+10 */$$

3. $$/* suma=\sum_{j=1}^{i} A[j] */$$
   $$\mathbf{i = i + 1;}$$
   $$/* suma=\sum_{j=1}^{i-1} A[j] */$$

4. $$/* 1 \le i+1 \le n \wedge j=5 */$$
   $$\mathbf{j = i + 1;}$$
   $$/* 1 \le j \le n */$$

5. $$/* z-x=y */$$
   $$\mathbf{z = z + x;}$$
   $$/* z=2x+y */$$

6. $$/* 1 \le i < n \wedge suma=\sum_{j=1}^{i} A[j] */$$
   $$\mathbf{suma = suma + A[i+1];}$$
   $$/* 1 \le i+1 \le n \wedge suma=\sum_{j=1}^{i+1} A[j] */$$

7. $$/* par(x) */$$
   $$\mathbf{x = x + 1;}$$
   $$/* impar(x) */$$

8. $$/* x=2^{y} */$$
   $$\mathbf{x = x * 2;}$$
   $$/* x=2^{(y+1)} */$$

Soluciones propuestas Verificación (I)

1. $/* y=5 */$
   $\boldsymbol{x=y*y+y;}$
   $/* x=30 */$

   1. $y=5 \rightarrow y*y+y=30$
   2. $/* y*y+y=30 */ \boldsymbol{x=y*y+y;} /* x=30 */ (\boldsymbol{AA})$
   3. $/* y=5 */ \boldsymbol{x=y*y+y;} /* x=30 */ (\boldsymbol{RCN}\ \mathbf{1,2})$

2. $/* x=y+5 */$
   $\boldsymbol{z=x+5;}$
   $/* x=y+5 \wedge z=y+10 */$

   1. $x=y+5 \rightarrow x+5=y+5+5 \rightarrow x+5=y+10$
   2. $/* x+5=y+10 */ \boldsymbol{z=x+5;} /* z=y+10 \wedge x+5=y+10 */ (\boldsymbol{AA})$
   3. $z=y+10 \wedge x+5=y+10 \rightarrow x=y+5 \wedge z=y+10$
   4. $/* x=y+5 */ \boldsymbol{z=x+5;} /* x=y+5 \wedge z=y+10 */ (\boldsymbol{RCN}\ \mathbf{1,2,3})$

3. $/* suma=\sum_{j=1}^{i} A[j] */$
   $\boldsymbol{i=i+1;}$
   $/* suma=\sum_{j=1}^{i-1} A[j] */$

   1. $suma=\sum_{j=1}^{i} A[j] \rightarrow suma=\sum_{j=1}^{i+1-1} A[j]$
   2. $/* suma=\sum_{j=1}^{i+1-1} A[j] */ \boldsymbol{i=i+1;} /* suma=\sum_{j=1}^{i-1} A[j] */ (\boldsymbol{AA})$
   3. $/* suma=\sum_{j=1}^{i} A[j] */ \boldsymbol{i=i+1;} /* suma=\sum_{j=1}^{i-1} A[j] */ (\boldsymbol{RCN}\ \mathbf{1,2})$

4. $/* 1 \leq i+1 \leq n \wedge j=5 */$
   $\boldsymbol{j=i+1;}$
   $/* 1 \leq j \leq n */$

   1. $1 \leq i+1 \leq n \wedge j=5 \rightarrow 1 \leq i+1 \leq n$
   2. $/* 1 \leq i+1 \leq n */ \boldsymbol{j=i+1;} /* 1 \leq j \leq n */ (\boldsymbol{AA})$

3. $/*1 \le i+1 \le n \wedge j=5*/$ **`j=i+1;`** $/*1 \le j \le n*/ (\textbf{RCN 1,2})$

5.
$/*\ z-x=y\ */$
`z=z+x;`
$/*\ z=2x+y\ */$

1. $z-x=y \rightarrow z-x+2x=y+2x \rightarrow z+x=y+2x$
2. $/*\ z+x=y+2x\ */$ **`z=z+x;`** $/*\ z=y+2x\ */ (\textbf{AA})$
3. $z=y+2x \rightarrow z=2x+y$
4. $/*\ z-x=y\ */$ **`z=z+x;`** $/*\ z=2x+y\ */ (\textbf{RCN 1,2,3})$

6.
$$/*\ 1 \le i < n \wedge suma = \sum_{j=1}^{i} A[j]\ */$$
`suma=suma+A[i+1];`
$$/*\ 1 \le i+1 \le n \wedge suma = \sum_{j=1}^{i+1} A[j]\ */$$

$$(1 \le i < n \wedge suma = \sum_{j=1}^{i} A[j]) \rightarrow (1 \le i < n \wedge suma + A[i+1] = \sum_{j=1}^{i+1} A[j])$$

1. $$\rightarrow (1 < i+1 \le n \wedge suma + A[i+1] = \sum_{j=1}^{i+1} A[j])$$
   $$\rightarrow (1 \le i+1 \le n \wedge suma + A[i+1] = \sum_{j=1}^{i+1} A[j])$$

2. $$/*\ 1 \le i+1 \le n \wedge suma + A[i+1] = \sum_{j=1}^{i+1} A[j]\ */$$ **`suma=suma+A[i+1];`**
   $$/*\ 1 \le i+1 \le n \wedge suma = \sum_{j=1}^{i+1} A[j]\ */ (\textbf{AA})$$

3. $$/*\ 1 \le i < n \wedge suma = \sum_{j=1}^{i} A[j]\ */$$ **`suma=suma+A[i+1];`**
   $$/*\ 1 \le i+1 \le n \wedge suma = \sum_{j=1}^{i+1} A[j]\ */ (\textbf{RCN 1,2})$$

7.
$/*\ par(x)\ */$
`x=x+1;`
$/*\ impar(x)\ */$

1. $par(x) \rightarrow impar(x+1)$
2. $/*\ impar(x+1)\ */$ **`x=x+1;`** $/*\ impar(x)\ */ (\textbf{AA})$

3. /* $par(x)$ */ $\boldsymbol{x = x + 1;}$ /* $impar(x)$ */ $(\boldsymbol{RCN}\ \mathbf{1,2})$

8.
/* $x = 2^y$ */
$\boldsymbol{x = x * 2;}$
/* $x = 2^{(y+1)}$ */

1. $x = 2^y \rightarrow x * 2 = 2^y * 2 \rightarrow x * 2 = 2^{(y+1)}$
2. /* $x * 2 = 2^{(y+1)}$ */ $\boldsymbol{x = x * 2;}$ /* $x = 2^{(y+1)}$ */ $(\boldsymbol{AA})$
3. /* $x = 2^y$ */ $\boldsymbol{x = x * 2;}$ /* $x = 2^{(y+1)}$ */ $(\mathbf{1,2}\ \boldsymbol{RCN})$

3.6 Regla de la composición (RCP)

Con lo visto hasta el momento, sólo podemos verificar asignaciones sueltas. Si en vez de una única tuviésemos dos seguidas, nos haría falta alguna otra herramienta para hacerlo. Ése es el objetivo de la **Regla de la Composición** (RCP): unir partes simples del cálculo de Hoare para poder verificar programas compuestos por varias líneas.

$$\frac{/*\Phi*/\ P_1\ /*\Phi_1*/,\ /*\Phi_1*/\ P_2\ /*\Psi*/}{/*\Phi*/\ P_1 P_2\ /*\Psi*/}\ (\boldsymbol{RCP})$$

Esta regla nos permite unir partes de un programa ya verificadas siempre que tengan una aserción intermedia común (en el enunciado de la regla Φ_1)

Veamos con un ejemplo cómo se utiliza la **RCP**:

$$/*\ suma=\sum_{j=1}^{i} A[\,j\,]\wedge i<n\ */\equiv /*\Phi*/$$
$$\boldsymbol{i=i+1;}\equiv P_1$$
$$/*\Phi_1*/$$
$$\boldsymbol{suma=suma+A[i];}\equiv P_2$$
$$/*\ suma=\sum_{j=1}^{i} A[\,j\,]\wedge i\le n\ */\equiv /*\Psi*/$$

El programa está formado por dos instrucciones P_1 y P_2 , entre las cuales se cumplirán unas propiedades (Φ_1) que a priori desconoceremos. El procedimiento en estos casos es aplicar el AA para la primera asignación (si hace falta transformar alguna aserción, tendremos que aplicar la RCN), con lo que podremos saber qué se cumple en Φ_1 . Intentaremos aplicar el AA sobre la segunda instrucción y posteriormente aplicaremos la RCP para unir las dos partes.

Por lo tanto, el objetivo final es verificar que se cumplen $(\boldsymbol{a})/*\Phi*/\ P_1\ /*\Phi_1*/$ y $(\boldsymbol{b})\ /*\Phi_1*/\ P_2\ /*\Psi*/$ para poder aplicar la RCP.

Primero, trataremos de verificar $(\boldsymbol{a})/*\Phi*/\ P_1\ /*\Phi_1*/$. Para ello, primero formularemos Φ en función de *i+1*, luego aplicaremos AA y finalmente RCN:

1. $$suma=\sum_{j=1}^{i} A[\,j\,]\wedge i<n\rightarrow suma=(\sum_{j=1}^{i+1-1} A[\,j\,])\wedge i+1\le n$$

2. $$/*\ suma=(\sum_{j=1}^{i+1-1} A[\,j\,])\wedge i+1\le n\ */\ \boldsymbol{i=i+1;}\ /*\ suma=(\sum_{j=1}^{\boldsymbol{i}-1} A[\,j\,]\wedge i\le n)\ (\boldsymbol{AA})$$

3. $/* suma = \sum_{j=1}^{i} A[j] \wedge i < n */ \; \mathbf{i = i + 1;} \; /* suma = \sum_{j=1}^{i-1} A[j] \wedge i \leq n */ \; (1,2\, RCN)$

Llegados a este punto, ya habremos verificado la primera parte (a), y sabremos que la aserción intermedia será $\Phi_1 \equiv /* suma = \sum_{j=1}^{i-1} A[j] \wedge i \leq n */$. Partiremos de ésta para intentar llegar a verificar $(\boldsymbol{b}) \; /* \Phi_1 */ \; P_2 \; /* \Psi */$. Empezaremos poniendo Φ_1 en función de *suma + A[i]*. Igual que para la primera parte, luego aplicaremos AA y finalmente RCN:

4. $suma = \sum_{j=1}^{i-1} A[j] \wedge i \leq n \rightarrow suma + A[i] = \sum_{j=1}^{i} A[j] \wedge i \leq n$

5. $/* suma + A[i] = \sum_{j=1}^{i} A[j] \wedge i \leq n */ \; \mathbf{suma = suma + A[i];} \; /* suma = \sum_{j=1}^{i} A[j] \wedge i \leq n */ (\boldsymbol{AA})$

6. $/* suma = \sum_{j=1}^{i-1} A[j] \wedge i \leq n */ \; \mathbf{suma = suma + A[i];} \; /* suma = \sum_{j=1}^{i} A[j] \wedge i \leq n */ (\mathbf{4,5\, RCN})$

Ahora que ya tenemos verificados $(\boldsymbol{a}) /* \Phi */ \; P_1 \; /* \Phi_1 */$ y $(\boldsymbol{b}) /* \Phi_1 */ \; P_2 \; /* \Psi */$ en los pasos 3 y 6, respectivamente, del cálculo, podemos aplicar la RCP sobre ambas propiedades:

7. $/* suma = \sum_{j=1}^{i} A[j] \wedge i < n */$
$\mathbf{i = i + 1;}$
$\mathbf{suma = suma + A[i];}$
$/* suma = \sum_{j=1}^{i} A[j] \wedge i \leq n */ \; (\mathbf{3,6\, RCP})$

Ejercicios Verificación (II)

Verifica los siguientes programas:

1. $$/* x=a \wedge y=b */ \equiv \Phi$$
   $$\boldsymbol{x = x + y;}$$
   $$\boldsymbol{y = x - y;}$$
   $$\boldsymbol{x = x - y;}$$
   $$/* x=b \wedge y=a */ \equiv \Psi$$

2. $$/* z=p^k */ \equiv \Phi$$
   $$\boldsymbol{k = k + 1;}$$
   $$\boldsymbol{z = z * p;}$$
   $$/* z=p^k */ \equiv \Psi$$

3. $$/* x=a \wedge y=b */ \equiv \Phi$$
   $$\boldsymbol{x = x + y;}$$
   $$\boldsymbol{y = x * x;}$$
   $$/* x=a+b \wedge y=(a+b)^2 */ \equiv \Psi$$

Soluciones propuestas Verificación (II)

1.

$$
\begin{array}{l}
/*\, x=a \wedge y=b \,*/ \equiv \Phi \\
\boldsymbol{x = x + y\ ;} \equiv A_1 \\
\qquad\qquad /*\, \Phi_1 \,*/ \\
\boldsymbol{y = x - y\ ;} \equiv A_2 \\
\qquad\qquad /*\, \Phi_2 \,*/ \\
\boldsymbol{x = x - y\ ;} \equiv A_3 \\
/*\, x=b \wedge y=a \,*/ \equiv \Psi
\end{array}
$$

/* Φ */ A_1; /* Φ_1 */

1. $x=a \wedge y=b \rightarrow \boldsymbol{x+y=a+b \wedge y=b}$
2. $/*\, x+y=a+b \wedge y=b \,*/\ \boldsymbol{x = x + y\ ;}\ /*\, \boldsymbol{x=a+b} \wedge y=b \,*/ \equiv /*\, \Phi_1 \,*/ (\boldsymbol{AA})$
3. $/*\, x=a \wedge y=b \,*/\ \boldsymbol{x = x + y\ ;}\ /*\, x=a+b \wedge y=b \,*/ (\boldsymbol{RCN}\ \mathbf{1,2})$

   /* Φ_1 */ A_2; /* Φ_2 */
4. $x=a+b \wedge y=b \rightarrow \boldsymbol{x=a+b} \wedge \boldsymbol{x-y=(a+b)-b \rightarrow x=a+b \wedge x-y=a}$
5. $/*\, x=a+b \wedge x-y=a \,*/\ \boldsymbol{y = x - y\ ;}\ /*\, x=a+b \wedge \boldsymbol{y=a} \,*/ \equiv /*\, \Phi_2 \,*/ (\boldsymbol{AA})$
6. $/*\, x=a+b \wedge y=b \,*/\ \boldsymbol{y = y - x\ ;}\ /*\, x=a+b \wedge y=a \,*/ (\boldsymbol{RCN}\ \mathbf{4,5})$

   /* Φ_2 */ A_3; /* Ψ */
7. $x=a+b \wedge y=a \rightarrow x-y=(a+b)-a \wedge y=a \rightarrow \boldsymbol{x-y=b} \wedge \boldsymbol{y=a}$
8. $/*\, x-y=b \wedge y=a \,*/\ \boldsymbol{x = x - y\ ;}\ /*\, \boldsymbol{x=b} \wedge y=a \,*/ \equiv /*\, \Psi \,*/ (\boldsymbol{AA})$
9. $/*\, x=a+b \wedge y=a \,*/\ \boldsymbol{x = x - y\ ;}\ /*\, x=b \wedge y=a \,*/ (\boldsymbol{RCN}\ \mathbf{7,8})$
10. $/*\, \Phi \,*/ A_1; A_2; A_3;\ /*\, \Psi \,*/ (\boldsymbol{RCP}\ \mathbf{3,6,9})$

2.

$$
\begin{array}{l}
/*\, z=p^k \,*/ \equiv \Phi \\
\boldsymbol{k = k + 1\ ;} \equiv A_1; \\
\qquad\qquad /*\, \Phi_1 \,*/ \\
\boldsymbol{z = z * p\ ;} \equiv A_2; \\
/*\, z=p^k \,*/ \equiv \Psi
\end{array}
$$

/* Φ */ A_1; /* Φ_1 */

1. $z=p^k \rightarrow z*p=p^{(k+1)}$
2. $/*\, z*p=p^{(k+1)} \,*/\ \boldsymbol{k = k + 1\ ;}\ /*\, z*p=p^{\boldsymbol{k}} \,*/ \equiv /*\, \Phi_1 \,*/ (\boldsymbol{AA})$
3. $/*\, z=p^k \,*/\ \boldsymbol{k = k + 1\ ;}\ /*\, z*p=p^k \,*/ (\boldsymbol{RCN}\ \mathbf{1,2})$

   /* Φ_1 */ A_2; /* Ψ */

4. $/*\, z * p = p^k \,*/\ \boldsymbol{z = z * p\ ;}\ /*\, z = p^k \,*/\ (\boldsymbol{AA})$
5. $/*\, \Phi \,*/\ \boldsymbol{A_1\ ;\ A_2\ ;}\ /*\, \Psi \,*/\ (\boldsymbol{RCP}\ \mathbf{3,4})$

3.
$$/*\, x = a \wedge y = b \,*/ \equiv \Phi$$
$$\boldsymbol{x = x + y\ ;} \equiv A_1$$
$$/*\, \Phi_1 \,*/$$
$$\boldsymbol{y = x * x\ ;} \equiv A_2$$
$$/*\, x = a + b \wedge y = (a+b)^2 \,*/ \equiv \Psi$$

/* Φ */ A_1 ; /* Φ_1 */

1. $x = a \wedge y = b \rightarrow \boldsymbol{x + y = a + b} \wedge \boldsymbol{y = b}$
2. $/*\, x + y = a + b \wedge y = b \,*/\ \boldsymbol{x = x + y\ ;}\ /*\, \boldsymbol{x = a + b} \wedge y = b \,*/ \equiv /*\, \Phi_1 \,*/\ (\boldsymbol{AA})$
3. $/*\, x = a \wedge y = b \,*/\ \boldsymbol{x = x + y\ ;}\ /*\, x = a + b \wedge y = b \,*/\ (\boldsymbol{RCN}\ \mathbf{1,2})$

   /* Φ_1 */ A_2 ; /* Ψ */
4. $x = a + b \wedge y = b \rightarrow \boldsymbol{x = a + b} \wedge \boldsymbol{x * x = (a + b)^2}$
5. $/*\, x = a + b \wedge x * x = (a+b)^2 \,*/\ \boldsymbol{y = x * x\ ;}$
   $/*\, x = a + b \wedge \boldsymbol{y = (a+b)^2} \,*/ \equiv /*\, \Psi \,*/\ (\boldsymbol{AA})$
6. $/*\, x = a + b \wedge y = b \,*/\ \boldsymbol{y = x * x\ ;}$
   $/*\, x = a + b \wedge y = (a+b)^2 \,*/\ (\boldsymbol{RCN}\ \mathbf{4,5})$
7. $/*\, \Phi \,*/\ \boldsymbol{A_1\ ;\ A_2\ ;}\ /*\, \Psi \,*/\ (\boldsymbol{RCP}\ \mathbf{3,6})$

3.7 Regla Condicional (RCD)

Con las reglas anteriores podemos verificar programas compuestos sólo por asignaciones. Otras instrucciones necesarias para poder verificar programas completos son las condicionales.

Para verificar un programa como el siguiente:

$$
\begin{array}{l}
/*\Phi*/ \\
if\ (B) \\
\boldsymbol{P_{if}}; \\
else \\
\boldsymbol{P_{else}}; \\
/*\Psi*/
\end{array}
$$

Utilizaremos la Regla Condicional (RCD), que se enuncia de dos maneras diferentes dependiendo de si hay una rama *else* o no:

a) Cuando haya *else*:

$$\frac{\Phi \rightarrow \text{def(B)},\ /*\Phi \wedge B*/\ P_{if}\ /*\Psi*/,\ /*\Phi \wedge \neg B*/\ P_{else}\ /*\Psi*/}{/*\Phi*/\ \text{if(B)}\ P_{if}\ \text{else}\ \ P_{else}\ /*\Psi*/}\ (\boldsymbol{RCD})$$

b) Cuando no lo haya:

$$\frac{\Phi \rightarrow \text{def(B)},\ /*\Phi \wedge B*/\ P_{if}\ /*\Psi*/, (\Phi \wedge \neg B) \rightarrow \Psi}{/*\Phi*/\ \text{if(B)}\ P_{if}\ /*\Psi*/}\ (\boldsymbol{RCD})$$

Antes de ver con un ejemplo cómo aplicar la RCD, intentaremos entender intuitivamente el porqué de cada una de las propiedades a demostrar para verificar un *if*:

- $\Phi \rightarrow \text{def(B)}$: La condición a evaluar tiene que estar definida por la aserción anterior, es decir, tendremos que verificar que la evaluación de la condición *B* nunca dará un resultado indeterminado o error.

- $/*\Phi \wedge B*/\ P_{if}\ /*\Psi*/,$: Cuando antes del *if* se cumplan Φ y la condición *B*, es decir, cuando entremos en el *then*, tendremos que verificar que después de ejecutar P_{if} siempre se verificará Ψ .

- $/*\Phi \wedge \neg B*/\ P_{else}\ /*\Psi*/$: En caso de haber *else*, tendremos que verificar que después de ejecutar P_{else} (antes siempre se cumplirá $\Phi \wedge \neg B$).

- $(\Phi \wedge \neg B) \rightarrow \Psi$: Cuando no haya *else*, habrá que verificar que $\Phi \wedge \neg B$ (lo que

sabremos que se cumplirá ya que habremos entrado en el *else*) implica directamente Ψ .

Cabe destacar que la diferencia entre un if con o sin else estriba en que, mientras en el primer caso se ejecutará código (P_{else}), en el segundo no, por lo que Ψ se tendrá que poder deducir de las propiedades que se cumplirán en el else: $\Phi \wedge \neg B$.

Ejemplo 1 (sin *else*):

Pasemos a ver con un ejemplo en cómo se aplica la RCD. Veamos el cálculo de Hoare asociado a la verificación del programa:

$$\begin{array}{l} /*\ \text{true}\ */ \equiv /* \Phi */ \\ \textbf{\textit{if}}(x<0) \\ x=-x; \\ /* x \geq 0 */ \equiv /* \Psi */ \end{array}$$

En este caso, $\Phi \equiv true$ y $B \equiv x<0$. Sólo tenemos un *if*, por lo que el objetivo final del cálculo será aplicar la RCD (sin *else*). Primero tendremos que verificar cada una de las 3 propiedades necesarias para poderla aplicar: **(A)** $\Phi \rightarrow \mathbf{def}(B)$, **(B)**$/* \Phi \wedge B */\ P_{if}\ /* \Psi */$ y $(C)(\Phi \wedge \neg B) \rightarrow \Psi$ **:** .

$(A)\Phi \rightarrow \mathbf{def}(B)$: *x<0* no puede dar un resultado indeterminado o error, por lo que:

1. $true \rightarrow \text{def}(x<0)$

(B)$/* \Phi \wedge B */\ P_{if}\ /* \Psi */$**:** : Tenemos una asignación, por lo que transformaremos $\Phi \wedge B$ para poder aplicar AA, lo aplicaremos, volveremos a transformar el resultado obtenido y finalmente uniremos los 3 pasos gracias a la RCN.

2. $(true \wedge x<0) \rightarrow x<0 \rightarrow -x>0$

3. $/* -x>0 */\ x=-x;\ /* x>0 */\ (AA)$

4. $x>0 \rightarrow x \geq 0$

5. $/* true \wedge x<0 */\ x=-x;\ */\ x \geq 0 */\ (RCN\ 2,3,4)$

$(C)(\Phi \wedge \neg B) \rightarrow \Psi$: En este último paso deberemos demostrar que aunque no entremos en el *then*, siempre se acabará satisfaciendo la post-condición.

6. $true \wedge \neg(x<0) \rightarrow \neg(x<0) \rightarrow x \geq 0$

Ahora sólo nos falta aplicar RCD:

7. /* true */ **`if`** (**`x`** < **`0`**) **`x`** = − **`x`** **`;`** /* $x \geq 0$ */ (***RCD*** **1,5,6**)

Ejemplo 2 (con *else*):

El ejemplo anterior era un *if* sin rama *else*, veamos ahora el cálculo de Hoare para uno que sí la tiene:

$$
\begin{array}{l}
/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])| \,*/ \equiv /*\, \Phi \,*/ \\
\textbf{\textit{if}}\ (\textbf{\textit{A}}[\textbf{\textit{i}}] \ge 0)\ \textbf{\textit{sumaAbs}} = \textbf{\textit{sumaAbs}} + \textbf{\textit{A}}[\textbf{\textit{i}}];\\
\quad \textbf{\textit{else}}\ \textbf{\textit{sumaAbs}} = \textbf{\textit{sumaAbs}} - \textbf{\textit{A}}[\textbf{\textit{i}}];\\
/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ \equiv /*\, \Psi \,*/
\end{array}
$$

Como en este programa sí tenemos un *else*, para aplicar la RCD habrá que demostrar: **(A)** $\Phi \rightarrow \mathbf{def}(B)$, **(B)**$/*\, \Phi \wedge B \,*/\ P_{if}\ /*\, \Psi \,*/$ y $(C)/*\, \Phi \wedge \neg B \,*/\ P_{else}\ /*\, \Psi \,*/$**:** .

(A) $\Phi \rightarrow \mathbf{def}(B)$: Sabemos por Φ que $1 \le i \le n$, por lo que el acceso al array estará dentro de su rango y no se producirá error:

1. $(1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])|) \rightarrow \mathrm{def}(A[i] \ge 0)$

(B)$/*\, \Phi \wedge B \,*/\ P_{if}\ /*\, \Psi \,*/$: demostraremos que entrando en el *then*, después del *if* siempre se verificará Ψ .

2. $$(1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])|) \wedge A[i] \ge 0$$
$$\rightarrow 1 \le i \le n \wedge sumaAbs + A[i] = \sum_{j=1}^{i} |(A[j])|$$

3. $$/*\, 1 \le i \le n \wedge sumaAbs + A[i] = \sum_{j=1}^{i} |(A[j])| \,*/\ \textbf{\textit{sumaAbs}} = \textbf{\textit{sumaAbs}} + \textbf{\textit{A}}[\textbf{\textit{i}}];$$
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ (\textbf{\textit{AA}})$$

4. $$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])| \wedge A[i] \ge 0 \,*/\ \textbf{\textit{sumaAbs}} = \textbf{\textit{sumaAbs}} + \textbf{\textit{A}}[\textbf{\textit{i}}];$$
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ (\textbf{\textit{RCN}}\ \mathbf{2,3})$$

$(\boldsymbol{C})$ **/*** $\boldsymbol{\Phi \wedge \neg B}$ ***/** $\boldsymbol{P_{else}}$ **/*** $\boldsymbol{\Psi}$ ***/:** : Ahora comprobaremos que entrando en el else se acaba verificando Ψ .

5.
$$(1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])|) \wedge \neg(A[i] \ge 0)$$
$$\rightarrow (1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])|) \wedge A[i] < 0$$
$$\rightarrow 1 \le i \le n \wedge sumaAbs + (-A[i]) = \sum_{j=1}^{i} |(A[j])|$$
$$\rightarrow 1 \le i \le n \wedge sumaAbs - A[i] = \sum_{j=1}^{i} |(A[j])|$$

6.
$$/*\, 1 \le i \le n \wedge sumaAbs - A[i] = \sum_{j=1}^{i} |(A[j])| \,*/ \ \textbf{sumaAbs} = \textbf{sumaAbs} - \textbf{A}[\textbf{i}];$$
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ \ (\boldsymbol{AA})$$

7.
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])| \wedge \neg(A[i] \ge 0) \,*/ \ \textbf{sumaAbs} = \textbf{sumaAbs} - \textbf{A}[\textbf{i}];$$
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ \ (\boldsymbol{RCN}\ \mathbf{5,6})$$

Finalmente, aplicamos la RCD a las propiedades 1, 4 y 7.

8.
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i-1} |(A[j])| \,*/$$
$$\textbf{if}\ (\textbf{A}[\textbf{i}] \ge \textbf{0})\ \textbf{sumaAbs} = \textbf{sumaAbs} + \textbf{A}[\textbf{i}];$$
$$\textbf{else}\ \textbf{sumaAbs} = \textbf{sumaAbs} - \textbf{A}[\textbf{i}];$$
$$/*\, 1 \le i \le n \wedge sumaAbs = \sum_{j=1}^{i} |(A[j])| \,*/ \ (\boldsymbol{RCD}\ \mathbf{1,4,7})$$

3.8 Descomposición de programas

Cuando trabajamos con programas de longitud y complejidad considerable, resulta difícil realizar directamente el cálculo de Hoare, porque cuantas más instrucciones tiene un programa, mayor número de líneas tendrá el cálculo de Hoare correspondiente.

Por ello, se recomienda hacer una *descomposición del programa* previa a su verificación, por lo menos, mientras no se domine la verificación perfectamente.

Así, lo primero que haremos será escribir el programa en forma esquemática, dando nombres a cada una de las instrucciones, expresiones y aserciones. Por ejemplo, en el siguiente programa:

```
/*x=a*y*/
if (y>0)
{
 x=x-a;
 y=y-1;
}
else
{
 x=x+a;
 y=y+1;
}
/*x=a*y*/
```

Lo escribiremos de forma esquemática (en esta fase sólo nos importará la estructura del programa) dando nombres a cada una de las partes del mismo. Aunque no es relevante de qué manera los nombremos, utilizaremos la siguiente nomenclatura para facilitar la lectura del esquema: Φ (phi) para la precondición, Ψ (psi) para la postcondición, Φ_i para las aserciones intermedias, B_i para las expresiones condicionales y A_i para las asignaciones.

Siguiendo con el ejemplo, su forma esquemática sería:

```
/*Φ*/
if (B)
{
  A1;
/*Φ1*/
  A2;
}
else
{
  A3;
/*Φ2*/
  A4;
}
/*Ψ*/
```

Como el programa está compuesto por un único *if*, la última regla de inferencia que utilizaremos será la RCD, por lo que tendremos que verificar $(\boldsymbol{A})\Phi \rightarrow \text{def}(B)$, $(\boldsymbol{B})/*\Phi \wedge B*/\ A_1, A_2\ /*\Psi*/$ y $(\boldsymbol{C})/*\Phi \wedge \neg B*/\ A_3, A_4 /*\Psi*/$. De estas dos propiedades, dos (B y C) se pueden descomponer en otras dos porque están compuestas por dos asignaciones, con lo que para verificar B, tendremos que verificar primero $(\boldsymbol{B.1})/*\Phi \wedge B*/\ A_1;/*\Phi_1*/$ y $(\boldsymbol{B.2})/*\Phi_1*/\ A_2;/*\Psi*/$. Para verificar C, a su vez, tendremos que verificar primeramente $(\boldsymbol{C.1})/*\Phi \wedge \neg B*/\ A_3;/*\Phi_2*/$ y $(\boldsymbol{C.2})/*\Phi_2*/\ A_4;/*\Psi*/$.

La descomposición del programa será entonces la siguiente:

$(\boldsymbol{A})\Phi \rightarrow \text{def}(B)$
$(\boldsymbol{B})/*\Phi \wedge B*/\ A_1; A_2; /*\Psi*/$
$(\boldsymbol{B.1})/*\Phi \wedge B*/ A_1; /*\Phi_1*/$
$(\boldsymbol{B.2})/*\Phi_1*/ A_2; /*\Psi*/$
$(\boldsymbol{C})/*\Phi \wedge \neg B*/ A_3; A_4; /*\Psi*/$
$(\boldsymbol{C.1})/*\Phi \wedge \neg B*/ A_3; /*\Phi_2*/$
$(\boldsymbol{C.2})/*\Phi_2*/ A_4; /*\Psi*/$

Una vez tengamos el programa descompuesto en partes simples (verificaciones de una sola asignación o implicaciones lógicas), empezaremos por las hojas del árbol de arriba a abajo. En el ejemplo, primero verificaríamos *A*, luego *B.1* y *B.2* para poderlas unir mediante la RCP y verificar *B*, y luego *C.1* y *C.2* que uniríamos mediante la RCP. Finalmente aplicaríamos la RCD sobre *A*, *B* y *C*.

Ejercicios Verificación (III)

Verifica los siguientes programas:

1.
```
/*true*/
if(!par(x))
  x=x+1;
/* par(x)*/
```

2.
```
/*x=a*y*/
if(y>0)
{
  x=x-a;
  y=y-1;
}
else
{
  x=x+a;
  y=y+1;
}
/*x=a*y*/
```

3.
```
/*numPares=ℵ i(1≤i<k∧i mod 2=0)*/
if(k%2==0)
  numPares=numPares+1;
k=k+1;
/*numPares=ℵ i(1≤i<k∧i mod 2=0)*/
```

Descompón los siguientes programas:

4.
```
/*Φ*/
if(B)
  A1;
else
{
  A2;
  A3;
}
/*Ψ*/
```

5.
```
/*Φ*/
A1;
if(B)
  A2;
A3;
/*Ψ*/
```

Soluciones propuestas Verificación (III)

1.

$$
\begin{array}{l}
/*\,true\,*/ \equiv /*\,\Phi\,*/ \\
\texttt{if (!par(x))} \equiv B \\
\quad \texttt{x=x+1;} \equiv A_1 \\
/*\,par(x)\,*/ \equiv /*\,\Psi\,*/
\end{array}
$$

$(\boldsymbol{A})\,\Phi \rightarrow \textbf{def}\,(\boldsymbol{B}) \equiv true \rightarrow \mathrm{def}\,(!\,par\,(x))$

1. $true \rightarrow (!\,par\,(x))$

   $(\boldsymbol{B})\;/*\,\Phi \wedge B\,*/\;A_1\;/*\,\Psi\,*/ \equiv /*\,true \wedge !\,par\,(x)\,*/\;x = x+1\,;\,/*\,\Psi\,*/$

2. $true \wedge !\,par\,(x) \rightarrow par\,(x+1)$
3. $/*\,par\,(x+1)\,*/\;\texttt{x=x+1;}\;/*\,par\,(x)\,*/\;(\boldsymbol{AA})$
4. $/*\,true \wedge !\,par\,(x)\,*/\;\texttt{x=x+1;}\;/*\,par\,(x)\,*/\;(\boldsymbol{RCN}\;\mathbf{2,3})$

   $(\boldsymbol{B})\,\Phi \wedge \neg B \rightarrow \Psi \equiv true \wedge \neg(\neg(\,par\,(x))) \rightarrow par\,(x)$

5. $true \wedge \neg(\neg(\,par\,(x))) \rightarrow par\,(x)$
6. $/*\,\Phi\,*/\;\textbf{if}\;(\boldsymbol{B})\;A_1\texttt{;}\;/*\,\Psi\,*/\;(\boldsymbol{RCD}\;\mathbf{1,4,5})$

2.

```
/* x = a * y */
if (y > 0)
{
  x = x - a;
  y = y - 1;
}
else
{
  x = x + a;
  y = y + 1;
}
/* x = a * y */
```

La descomposición del programa será la siguiente:

$$
\begin{array}{l}
(\boldsymbol{A})\,\Phi \rightarrow \mathrm{def}\,(B) \\
(\boldsymbol{B})\;/*\,\Phi \wedge B\,*/\;A_1\texttt{;}\;A_2\texttt{;}\;/*\,\Psi\,*/ \\
\quad (\boldsymbol{B.1})\;/*\,\Phi \wedge B\,*/\;A_1\texttt{;}\;/*\,\Phi_1\,*/ \\
\quad (\boldsymbol{B.2})\;/*\,\Phi_1\,*/\;A_2\texttt{;}\;/*\,\Psi\,*/ \\
(\boldsymbol{C})\;/*\,\Phi \wedge \neg B\,*/\;A_3\texttt{;}\;A_4\texttt{;}\;/*\,\Psi\,*/ \\
\quad (\boldsymbol{C.1})\;/*\,\Phi \wedge \neg B\,*/\;A_3\texttt{;}\;/*\,\Phi_2\,*/ \\
\quad (\boldsymbol{C.2})\;/*\,\Phi_2\,*/\;A_4\texttt{;}\;/*\,\Psi\,*/
\end{array}
$$

$(\boldsymbol{A})\Phi \rightarrow \textbf{def}\,(\boldsymbol{B})$

1. $x = a * y \rightarrow \text{def}\,(y > 0)$

   $(\boldsymbol{B.1})$ /* $\Phi \wedge B$ */ **A1;** /* Φ_1 */

2. $x = a * y \wedge y > 0 \rightarrow x - a = (a * y) - a \rightarrow x - a = a * (y - 1)$
3. /* $x - a = a * (y - 1)$ */ **x = x - a;** /* $x = a * (y - 1)$ */ $\equiv \Phi_1(AA)$
4. /* $\Phi \wedge B$ */ **A1;** /* Φ_1 */ (**RCN 2,3**)

   $(\boldsymbol{B.2})$ /* Φ_1 */ **A2;** /* Ψ */

5. /* $x = a * (y - 1)$ */ **y = y - 1;** /* $x = a * y$ */ (AA)

   $(\boldsymbol{B})$ /* $\Phi \wedge B$ */ **A1; A2;** /* Ψ */

6. /* $\Phi \wedge B$ */ **A1; A2;** /* Ψ */ (**RCP 4,5**)

   $(\boldsymbol{C.1})$ /* $\Phi \wedge \neg B$ */ **A3;** /* Φ_2 */

7. $x = a * y \wedge y \leq 0 \rightarrow x + a = (a * y) + a \rightarrow x + a = a * (y + 1)$
8. /* $x + a = a * (y + 1)$ */ **x = x + a;** /* $x = a * (y + 1)$ */ ¿ $\Phi_2(AA)$
9. /* $\Phi \wedge \neg B$ */ **A3;** /* Φ_2 */ (**RCN 7,8**)

   $(\boldsymbol{C.2})$ /* Φ_2 */ **A4;** /* Ψ */

10. /* $x = a * (y + 1)$ */ **y = y + 1;** /* $x = a * y$ */ (AA)

    $(\boldsymbol{C})$ /* $\Phi \wedge \neg B$ */ **A3; A4;** /* Ψ */

11. /* $\Phi \wedge \neg B$ */ **A3; A4;** /* Ψ */ (**RCP 9,10**)
12. /* Φ */ **if (B) { A1; A2; } else { A3; A4; }** /* Ψ */ (**RCD 1,6,11**)

3.
```
/* numPares = N i (1 ≤ i < k ∧ i mod 2 = 0) */
if (k%2==0)
  numPares = numPares + 1; ≡ A1
k = k + 1; ≡ A2
/* numPares = N i (1 ≤ i < k ∧ i mod 2 = 0) */
```

La descomposición del programa será la siguiente:

$(\boldsymbol{A})$ /* Φ */ **if** (B) A_1; /* Φ_1 */
$(\boldsymbol{A.1})\ \Phi \rightarrow \text{def}\,(B)$
$(\boldsymbol{A.2})$ /* $\Phi \wedge B$ */ A_1; /* Φ_1 */
$(\boldsymbol{A.3})\ \Phi \wedge \neg B \rightarrow \Phi_1$
$(\boldsymbol{B})$ /* Φ_1 */ A_2; /* Ψ */

$(\boldsymbol{A.1})\ \Phi \rightarrow \textbf{def}\,(\boldsymbol{B})$

1. $numPares = \aleph\, i\,(1 \leq i < k \wedge i \bmod 2 = 0) \rightarrow \text{def}\,(k \bmod 2 = 0)$

$(A.2)$ /* $\Phi \wedge B$ */ **A₁;** /* Φ_1 */

2. $numPares = \aleph\, i(1 \le i < k \wedge i \bmod 2 = 0) \wedge k \bmod 2 = 0$
$\rightarrow numPares + 1 = \aleph\, i(\mathbf{1} \le \mathbf{i} \le \mathbf{k} \wedge i \bmod 2 = 0)$

3. /* $numPares + 1 = \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$ */ **numPares = numPares + 1;**
/* **numPares** $= \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$ */ $\equiv \Phi_1(AA)$

4. /* $numPares = \aleph\, i(1 \le i < k \wedge i \bmod 2 = 0) \wedge k \bmod 2 = 0$ */
numPares = numPares + 1;
/* $numPares = \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$ */ $(RCN\ 2,3)$

$(A.3)\ \Phi \wedge \neg B \rightarrow \Phi_1$

5. $numPares = \aleph\, i(1 \le i < k \wedge i \bmod 2 = 0) \wedge \neg(k \bmod 2 = 0)$
$\rightarrow numPares = \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$

(A) /* Φ */ **if** (B) **A₁;** /* Φ_1 */

6. /* Φ */ **if** (B) **A₁;** /* Φ_1 */ $(RCD\ 1,4,5)$

(B) /* Φ_1 */ **A₂;** /* Ψ */

7. $numPares = \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$
$\rightarrow numPares = \aleph\, i(\mathbf{1} \le \mathbf{i} < \mathbf{k} + \mathbf{1} \wedge i \bmod 2 = 0)$

8. /* $numPares = \aleph\, i(1 \le i < k + 1 \wedge i \bmod 2 = 0)$ */ **k = k + 1;**
/* $numPares = \aleph\, i(\mathbf{1} \le \mathbf{i} < \mathbf{k} \wedge i \bmod 2 = 0)$ */ (AA)

9. /* $numPares = \aleph\, i(1 \le i \le k \wedge i \bmod 2 = 0)$ */ **k = k + 1;**
/* $numPares = \aleph\, i(1 \le i < k \wedge i \bmod 2 = 0)$ */ $(RCN\ 7,8)$

(A) /* Φ */ **if** (B) **A₁; A₂;** /* Ψ */

10. /* Φ */ **if** (B) **A₁; A₂;** /* Ψ */ $(RCP\ 6,9)$

4.
```
/* Φ */
if (B)
  A1;
else
{
  A2;
  A3;
}
/* Ψ */
```

$(\boldsymbol{A})\ \Phi \rightarrow \mathrm{def}(B)$
$(\boldsymbol{B})\ /*\Phi \wedge B*/\ A_1;\ /*\Psi*/$
$(\boldsymbol{C})\ /*\Phi \wedge \neg B*/\ A_2;A_3;\ /*\Psi*/$
$(\boldsymbol{C.1})\ /*\Phi \wedge \neg B*/\ A_2;\ /*\Phi_1*/$
$(\boldsymbol{C.2})\ /*\Phi_1*/\ A_3;\ /*\Psi*/$

5.

```
/*Φ*/
A1;
if (B)
  A2;
A3;
/*Ψ*/
```

$(\boldsymbol{A})\ /*\Phi*/\ A_1;\ /*\Phi_1*/$
$(\boldsymbol{B})\ /*\Phi_1*/\ \texttt{if}\ (B)\ A_2;\ /*\Phi_2*/$
$(\boldsymbol{B.1})\ \Phi_1 \rightarrow \mathrm{def}(B)$
$(\boldsymbol{B.2})\ /*\Phi_1 \wedge B*/\ A_2;\ /*\Phi_2*/$
$(\boldsymbol{B.3})\ \Phi_1 \wedge \neg B \rightarrow \Phi_2$
$(\boldsymbol{C})\ /*\Phi_2*/\ A_3;\ /*\Psi*/$

3.9 Iteraciones

Una iteración realiza una **labor** a base de repetir un conjunto de instrucciones que:

- **Modifica** el valor de las variables
- Conserva alguna **relación** entre ellas, una propiedad

Antes de enunciar la regla de inferencia que nos permitirá verificar iteraciones como ésta:

$$
\begin{array}{l}
/*\,\Phi\,*/ \\
while\,(B) \\
\quad \mathbf{I};\\
/*\,\Psi\,*/
\end{array}
$$

Veremos cuáles son esas propiedades que se mantienen en todas las iteraciones, el invariante.

3.9.1 Concepto de Invariante

Un **invariante** es una propiedad, un predicado que no cambia, que se cumple antes (y al final) de cada ejecución de una iteración. No se trata de cualquier propiedad que se conserve sino de aquélla que *caracterice y establezca la semántica del bucle y su efecto.*

Una vez deducido el invariante (INV), las aserciones intermedias serán las siguientes:

```
/*Φ*/
          /* INV */
while(B)
{
    /* INV ∧ B */
I;
          /* INV */
}
   /* INV ∧ ¬B */
/*Ψ*/
```

El invariante, como se puede ver en la figura superior, se cumplirá antes del bucle, al inicio de cada iteración (donde además se cumplirá la condición B), al final de cada iteración y después del bucle (donde además se cumplirá la negación de B).

Veamos ahora unos ejemplos de bucles y sus respectivos invariantes.

Ejemplo 1:

Examinando el siguiente programa:

$$/*x=a \wedge y=b \geq 0*/ \equiv /*\Phi*/$$
```
while(y!=0)
{
        x=x+1;
        y=y-1;
{
```
$$/*x=a+b*/ \equiv /*\Psi*/$$

Podemos deducir que la **principal propiedad** que se mantiene en todas las iteraciones será que **la suma de *x* e *y* será constante**, ya que en el bucle incrementaremos en uno *x* y decrementaremos en uno *y*, es decir, le sumamos a una variable lo que le restamos a la otra. Por lo tanto, siempre se cumplirá $\boldsymbol{x+y=a+b}$.

Además, podremos saber cuál será el **rango de valores que tomará *y***. Inicialmente, tendrá un valor *b* positivo. En cada iteración se decrementará en uno, y se seguirá iterando hasta que *y* valga 0. Gracias a esto, podremos afirmar que siempre se cumplirá $\boldsymbol{y \geq 0}$. $y>0$ no sería suficiente, ya que el invariante también se tiene que cumplir cuando salgamos del bucle, y se saldrá cuando *y* valga 0.

Podríamos intentar delimitar el rango de valores de x, pero sólo nos interesará el de y, ya que es quien nos ayudará (por aparecer en la condición *B*) a deducir si el bucle terminará en un número finito de pasos (*terminación*, lo veremos más adelante).

Por lo tanto, el invariante será el siguiente:

$$/*\ \text{INV}\ */ \equiv /*x+y=a+b \wedge b \geq y \geq 0*/$$

Aplicándolo al esquema anteriormente presentado, sabremos que siempre se cumplirá:

$$/*x=a \wedge y=b \geq 0*/ \equiv /*\Phi*/$$
$$/*x+y=a+b \wedge y \geq 0*/ \equiv /*INV*/$$
```
while(y!=0)
{
```
$$/*x+y=a+b \wedge y \geq 0 \wedge y \neq 0*/ \equiv /*\ \text{INV} \wedge B*/$$
```
        x=x+1;
        y=y-1;
```
$$/*x+y=a+b \wedge y \geq 0*/ \equiv /*\ \text{INV}\ */$$
```
}
```
$$/*x+y=a+b \wedge y \geq 0 \wedge y=0*/ \equiv /*\ \text{INV} \wedge \neg B*/$$
$$/*x=a+b*/ \equiv /*\Psi*/$$

Ejemplo 2:

Examinando el siguiente programa:

$$/* suma = 0 \wedge k = 1 */ \equiv /* \Phi */$$
$$\textbf{while}(k \leq n)$$
$$\{$$
$$suma = suma + A[k];$$
$$k = k + 1;$$
$$\{$$
$$/* suma = \sum_{i=1}^{n} A[i] */ \equiv /* \Psi */$$

Por una parte, podremos deducir que en *suma* tendremos siempre la suma de los *k* primeros elementos del vector. Como, dentro del bucle, primero se suma el *k*-ésimo elemento y después se incrementa k, al inicio del bucle tendremos la suma hasta *k-1*. Siempre se cumplirá por lo tanto $\boldsymbol{suma = \sum_{i=1}^{k-1} A[i]}$. Es importante asegurarnos que la propiedad también se cumpla antes de entrar en el bucle. Como *k* es 1 inicialmente, si sustituímos *k* por su valor inicial en el invariante, el sumatorio tendrá *dominio vacío* (desde 1 hasta 0) por lo que suma deberá ser 0, y eso lo sabemos por la precondición.

Por otra, nos interesará otra vez asegurarnos de que se saldrá del bucle en un número finito de pasos, por lo que trataremos de delimitar el rango de *k*, que será la variable que determinará si seguimos o no iterando. Inicialmente, k valdrá 1, cada vuelta se incrementará en uno, y se saldrá cuando k sea mayor que n (cuando valga n+1). Por lo tanto, el rango de valores que tomará k será [1..n+1], que expresado en LPO será $\boldsymbol{1 \leq k \leq n+1}$.

El invariante será pues:

$$/* INV */ \equiv /* suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 */$$

Ejemplo 3:

Cambiando el orden en que se hacen las sumas, tendremos éste otro programa:

$$/* suma = 0 \wedge k = 0 */ \equiv /* \Phi */$$
$$\textbf{while}(k < n)$$
$$\{$$
$$k = k + 1;$$
$$suma = suma + A[k];$$
$$\{$$
$$/* suma = \sum_{i=1}^{n} A[i] */ \equiv /* \Psi */$$

En este otro ejemplo, como primero se incrementa el índice y luego se suma, tendremos en la variable *suma*, al inicio de cada iteración, la suma de los *k* primeros elementos, por lo que siempre se cumplirá $\boldsymbol{suma} = \sum_{i=1}^{k} A[i]$. Igual que en el caso anterior, comprobaremos que también se cumpla antes de entrar en el bucle. En este caso, *k* será 0, por lo que el sumatorio (desde 1 hasta *k*) tendrá dominio vacío, y como *suma* será 0, se cumplirá el invariante.

Respecto al rango de k, podremos afirmar que siempre estará en [0..n], ya que inicialmente es 0, cada vuelta se incrementa en uno y cuando se salga del bucle valdrá n, por lo que $\boldsymbol{0 \le k \le n}$.

El invariante será por lo tanto:

$$/* INV */ \equiv /* suma = \sum_{i=1}^{k} A[i] \wedge 0 \le k \le n */$$

3.9.2 Regla del While (RWH)

Para verificar la corrección parcial de un programa con iteraciones como la siguiente:

$$
\begin{array}{l}
/*\,\Phi\,*/ \\
while\,(B) \\
\quad \boldsymbol{I}\,; \\
/*\,\Psi\,*/
\end{array}
$$

Utilizaremos la Regla del While (RWH), que se enuncia de la siguiente manera:

$$\frac{\Phi \rightarrow INV\ ,\ INV \rightarrow \text{def(B)},\ /*\ INV \wedge B\ */\ I\ /*\ INV\ */,\ \ INV \wedge \neg B \rightarrow \Psi}{/*\,\Phi\,*/\ \text{while (B)}\ I\ ;/*\,\Psi\,*/}\ (\boldsymbol{RWH})$$

Al igual que con la RCD, trataremos de entender intuitivamente qué significa cada una de las propiedades necesarias para aplicar la RWH:

- $\Phi \rightarrow INV$: Como hemos dicho, el invariante se cumplirá antes de entrar en el bucle por primera vez, por lo que la aserción anterior al bucle tiene que implicar que se cumpla.

- $INV \rightarrow \text{def}\,(B)$: Antes de evaluar la condición (antes de entrar en el bucle y al final de cada iteración) siempre se cumplirá el invariante, por lo que tiene que definir *B*, tiene que garantizar que la evaluación de *B* no pueda dar un resultado inesperado o error.

- $/*\ INV \wedge B\ */\ I\ /*\ INV\ */$: Antes de cada iteración se cumplirán tanto el invariante como la condición *B*, por lo que tendremos que verificar que después de ejecutar el cuerpo del bucle (*I*), siempre se verifique el invariante al final del mismo.

- $INV \wedge \neg B \rightarrow \Psi$: Cuando la evaluación de la condición B dé falso, se tendrá que verificar Ψ. Al evaluar la condición siempre se cumplirá el invariante y, la última vez que se evalúe, retornará falso. Sabiendo esto, tenemos que deducir que siempre se satisfará la post Ψ.

Veamos ahora unos ejemplos de cómo aplicar la RWH.

Ejemplo 1:

Para el siguiente programa:

$$
\begin{array}{l}
/*\, x>1 \wedge y=1 \,*/ \\
\textbf{while}(y*y<x) \\
\{ \\
\qquad y=y+1; \\
\} \\
/*\, \forall\, k(1 \le k<y \rightarrow k*k<x) \,*/
\end{array}
$$

Primero trataremos de determinar cuál es el invariante. Si estudiamos el programa veremos que se irá incrementando la variable *y* en uno hasta que *y*y* sea igual a *x*. Es decir, al final del bucle tendremos en *y* el menor número que cumplirá $y*y \ge x$.

Por una parte, sabremos que si seguimos en el bucle, siempre se cumplirá que ninguno de los valores que haya tenido *y* cumplirá $y*y \ge x$, porque de otra manera, ya habríamos salido del bucle. Por lo tanto, siempre se cumplirá $\forall\, k(1 \le k<y \rightarrow k*k<x)$, desde la primera hasta la última vuelta.

En este caso, a diferencia de los ejemplos sobre invariantes, no nos resultará necesario incluir información sobre el rango de *y* (la variable que determinará el final del bucle) ya que la condición no puede dar un resultado indeterminado.

Por lo tanto, el invariante será el siguiente:

$$/*\, INV \,*/ \equiv \forall\, k(1 \le k<y \rightarrow k*k<x)$$

Ahora conoceremos algunas de las aserciones intermedias del programa:

$$
\begin{array}{l}
/*\, \Phi \,*/ \equiv /*\, x>1 \wedge y=1 \,*/ \\
\qquad\qquad /*\, INV \,*/ \equiv /*\, \forall\, k(1 \le k<y \rightarrow k*k<x) \,*/ \\
\textbf{while}(y*y<x) \\
\{ \\
\quad /*\, INV \wedge B \,*/ \equiv /*\, \forall\, k(1 \le k<y \rightarrow k*k<x) \wedge y*y<x \,*/ \\
y=y+1; \\
\qquad\qquad /*\, INV \,*/ \equiv /*\, \forall\, k(1 \le k<y-1 \rightarrow k*k<x) \,*/ \\
\} \\
/*\, INV \wedge \neg B \,*/ \equiv /*\, \forall\, k(1 \le k<y \rightarrow k*k<x) \wedge y*y \ge x \,*/ \\
/*\, \Psi \,*/ \equiv /*\, \forall\, k(1 \le k<y \rightarrow k*k<x) \,*/
\end{array}
$$

Al ser el programa un único while, antes de aplicar la RWH, deberemos verificar únicamente las cuatro propiedades necesarias: $(A) \Phi \rightarrow INV$, $(B)\, INV \rightarrow \mathrm{def}\,(B)$, $(C) /*\, INV \wedge B \,*/\, I\, /*\, INV \,*/$ y $(D)\, INV \wedge \neg B \rightarrow \Psi$.

$(\boldsymbol{A})\ \boldsymbol{\Phi} \rightarrow \boldsymbol{INV}$

1. $x>1 \wedge y=1 \rightarrow \forall\, k\,(1 \le k < y \rightarrow k*k<x)$ ya que el dominio $1 \le k < 1$ es vacío.

$(\boldsymbol{B})\ \boldsymbol{INV} \rightarrow \mathbf{def}\,(\boldsymbol{B})$

2. $\forall\, k\,(1 \le k < y \rightarrow k*k<x) \rightarrow \mathrm{def}\,(y*y<x)$

$(\boldsymbol{C})$ /* $\boldsymbol{INV} \wedge \boldsymbol{B}$ */ $\boldsymbol{A_1}$; /* $\boldsymbol{INV}$ */

3. $\forall\, k\,(1 \le k < y \rightarrow k*k<x) \wedge y*y<x$
$\rightarrow \forall\, k\,(\mathbf{1} \le \boldsymbol{k} \le \boldsymbol{y} \rightarrow k*k<x)$
$\rightarrow \forall\, k\,(\mathbf{1} \le \boldsymbol{k} < \boldsymbol{y}+\mathbf{1} \rightarrow k*k<x)$

4. /* $\forall\, k\,(1 \le k < y+1 \rightarrow k*k<x)$ */ $\boldsymbol{y} = \boldsymbol{y}+\mathbf{1}$;
/* $\forall\, k\,(1 \le k < \boldsymbol{y} \rightarrow k*k<x)$ */ $\equiv$ /* Ψ */ $(\boldsymbol{AA})$

5. /* $\forall\, k\,(1 \le k < y \rightarrow k*k<x) \wedge y*y<x$ */ $\boldsymbol{y} = \boldsymbol{y}+\mathbf{1}$;
/* $\forall\, k\,(1 \le k < y \rightarrow k*k<x)$ */ $\equiv$ /* Ψ */ $(\boldsymbol{RCN}\ \mathbf{3,4})$

$(\boldsymbol{D})\ \boldsymbol{INV} \wedge \neg \boldsymbol{B} \rightarrow \boldsymbol{\Psi}$

6. $\forall\, k\,(1 \le k < y \rightarrow k*k<x) \wedge \neg(y*y<x)$
$\rightarrow \forall\, k\,(1 \le k < y \rightarrow k*k<x) \wedge y*y \ge x \equiv$ /* *PSI* */

Finalmente, aplicamos la RWH sobre las cuatro propiedades:

7. /* Φ */ [**while** $(\boldsymbol{B})\ \boldsymbol{A_1}$;] /* Ψ */ $(\boldsymbol{RWH}\ \mathbf{1,2,5,6})$

Ejemplo 2:

Como segundo ejemplo de RWH reutilizaremos uno de los utilizados para explicar los invariantes:

$$
\begin{array}{l}
/* \; suma = 0 \wedge k = 1 \; */ \equiv /* \; \Phi \; */ \\
\textbf{\textit{while}}\,(\boldsymbol{k \leq n}) \\
\{ \\
\boldsymbol{suma = suma + A[k];} \\
\boldsymbol{k = k + 1;} \\
\{ \\
/* \; suma = \sum_{i=1}^{n} A[i] \; */ \equiv /* \; \Psi \; */
\end{array}
$$

Como ya dedujésemos en su momento, el invariante será:

$$/* \; INV \; */ \equiv /* \; suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \; */$$

Al estar compuesto el cuerpo de la iteración (*I*) por dos instrucciones, la descomposición del programa no será tan obvia como en el ejemplo anterior:

$$
\begin{array}{l}
(\boldsymbol{A}) \; \Phi \rightarrow INV \\
(\boldsymbol{B}) \; INV \rightarrow \mathrm{def}\,(B) \\
(\boldsymbol{C}) /* \; INV \wedge B \; */ A_1 ; A_2 ; \; /* \; INV \; */ \\
\quad (\boldsymbol{C.1}) /* \; INV \wedge B \; */ A_1 ; \; /* \; \Phi_1 \; */ \\
\quad (\boldsymbol{C.2}) /* \; \Phi_1 \; */ A_2 ; \; /* \; INV \; */ \\
(\boldsymbol{D}) \; INV \wedge \neg B \rightarrow \Psi
\end{array}
$$

$(\boldsymbol{A})\,\boldsymbol{\Phi} \rightarrow \boldsymbol{INV}$

1. $suma = 0 \wedge k = 1 \rightarrow suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1$ ya que el sumatorio desde 1 hasta 1-1=0 es 0, que es el contenido de la variable *suma*.

$(\boldsymbol{B})\,\boldsymbol{INV} \rightarrow \mathbf{def}\,(\boldsymbol{B})$

2. $suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \rightarrow \mathrm{def}\,(k \leq n)$

$(\boldsymbol{C.1})$ /* $\boldsymbol{INV} \wedge \boldsymbol{B}$ */ $\boldsymbol{A_1}$; /* $\boldsymbol{\Phi_1}$ */

3. $suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \wedge (k \leq n) \rightarrow \boldsymbol{suma} + \boldsymbol{A}[\boldsymbol{k}] = \sum_{i=1}^{k} A[i] \wedge 1 \leq k \leq n$

4. /* $suma + A[k] = \sum_{i=1}^{k} A[i] \wedge 1 \leq k \leq n$ */ $\boldsymbol{suma} = \boldsymbol{suma} + \boldsymbol{A}[\boldsymbol{k}]$

   /* $\boldsymbol{suma} = \sum_{i=1}^{k} A[i] \wedge 1 \leq k \leq n$ */ $(\boldsymbol{AA})$

5. /* $suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \wedge (k \leq n)$ */ $\boldsymbol{suma} = \boldsymbol{suma} + \boldsymbol{A}[\boldsymbol{k}]$;

   /* $\boldsymbol{suma} = \sum_{i=1}^{k} A[i] \wedge 1 \leq k \leq n$ */ $(\boldsymbol{RCN}\ \mathbf{3,4})$

$(\boldsymbol{C.2})$ /* $\boldsymbol{\Phi_1}$ */ $\boldsymbol{A_2}$; /* $\boldsymbol{INV}$ */

6. $suma = \sum_{i=1}^{k} A[i] \wedge 1 \leq k \leq n \rightarrow suma = \sum_{i=1}^{k+1-1} A[i] \wedge 1 \leq k+1 \leq n+1$

7. /* $suma = \sum_{i=1}^{\boldsymbol{k+1}-1} A[i] \wedge 1 \leq k+1 \leq n+1$ */ $\boldsymbol{k} = \boldsymbol{k} + \mathbf{1}$;

   /* $suma = \sum_{i=1}^{\boldsymbol{k}-1} A[i] \wedge 1 \leq \boldsymbol{k} \leq n+1$ */ $(\boldsymbol{AA})$

8. /* $suma = \sum_{i=1}^{k} A[i] \wedge (k \leq n)$ */ $\boldsymbol{k} = \boldsymbol{k} + \mathbf{1}$;

   /* $suma = \sum_{i=1}^{k-1} A[i] \wedge (1 \leq k \leq n+1)$ */ $(\boldsymbol{RCN}\ \mathbf{6,7})$

$(\boldsymbol{C})$ /* $\boldsymbol{INV} \wedge \boldsymbol{B}$ */ $\boldsymbol{A_1; A_2;}$ /* $\boldsymbol{INV}$ */

9. /* $INV \wedge B$*/ $\boldsymbol{A_1; A_2;}$ /* INV*/ $(\boldsymbol{RCP}\ \mathbf{5,9})$

   $(\boldsymbol{D})\ \boldsymbol{INV} \wedge \neg \boldsymbol{B} \rightarrow \Psi$

10. $suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq \boldsymbol{k} \leq \boldsymbol{n}+\mathbf{1} \wedge \boldsymbol{k} > \boldsymbol{n} \rightarrow suma = \sum_{i=1}^{\boldsymbol{k}-\mathbf{1}} A[i] \wedge \boldsymbol{k} = \boldsymbol{n}+\mathbf{1} \rightarrow suma = \sum_{i=1}^{\boldsymbol{n}} A[i]$

11. /*Φ*/ $[\boldsymbol{while}(\boldsymbol{B})\{\boldsymbol{A_1; A_2;}\}]$ /*Ψ*/ $\boldsymbol{RWH}\ (\mathbf{1,2,10,11})$

3.9.3 Terminación

En los ejemplos anteriores, no acabábamos el cálculo de Hoare *verificando totalmente* los programas ($/*\Phi*/[\mathbf{while}(\mathbf{B})\{\mathbf{I;}\}]/*\Psi*/$), sino que los *verificábamos parcialmente* ($/*\Phi*/\mathbf{while}(\mathbf{B})\{\mathbf{I;}\}/*\Psi*/$). La principal diferencia entre los programas con bucles y los que no tienen consiste en que los primeros *pueden no acabar*. Puede ocurrir que algún bucle, como el del ejemplo siguiente, sea infinito y que nunca se llegue a salir del bucle:

$$
\begin{array}{l}
/*\ suma = 0 \wedge k = 1\ */ \equiv /*\ \Phi\ */ \\
\mathbf{while}(\mathbf{k} \leq \mathbf{n}) \\
\{ \\
\mathbf{suma} = \mathbf{suma} + \mathbf{A}[\mathbf{k}]\mathbf{;} \\
\{ \\
/*\ suma = \sum_{i=1}^{n} A[i]\ */ \equiv /*\ \Psi\ */
\end{array}
$$

En este ejemplo anterior no se actualiza el valor de *k* al final de cada vuelta por lo que será un **bucle infinito**.

Por lo tanto, en el caso de programas con bucles no será suficiente lo visto hasta el momento para verificar la corrección total del programa. Además hará falta asegurarse de que el programa terminará en un número finito de pasos ($\mathbf{Term}(\Phi, \mathbf{P})$).

Para ello, tendremos que elegir una **expresión cota(E)** que cumpla:

- $INV \wedge B \rightarrow \exists c(E > c)$. La expresión tiene una *cota inferior c*.
- $/*\ INV \wedge B \wedge E = v\ */\ \mathbf{I;}\ /*\ E < v\ */$. El valor devuelto por la expresión deberá decrecer en cada una de las iteraciones.
- **La expresión E elegida deberá estar formada por variables enteras y operaciones enteras**, ya que hay infinitos números reales entre cualesquiera dos números reales.

Una vez que hemos verificado la corrección parcial y estas dos propiedades para la expresión cota, podremos afirmar que el programa P es totalmente correcto ($/*\Phi*/\mathbf{[while}(\mathbf{B})\mathbf{\{I;\}]}/*\Psi*/$).

Nótese que la expresión cota elegida tiene que decrecer en cada iteración, por lo que deberá de dar una medida del número de iteraciones que quedan por realizar.

Ahora verificaremos la corrección total de los dos ejemplos utilizados para ilustrar la RWH.

Ejemplo 1:

$$
\begin{array}{l}
/*\, x>1 \wedge y=1 \,*/ \\
\textbf{while}(\boldsymbol{y*y<x}) \\
\{ \\
\boldsymbol{y=y+1;} \\
\} \\
/*\, \forall\, k\,(1\le k<y \rightarrow k*k<x)\,*/
\end{array}
$$

Primero debemos elegir la expresión cota. En este caso, lo que decrece entre las diferentes iteraciones será $x-y$, ya que *y* crecerá, decreciendo así el resultado de la expresión $\boldsymbol{E\equiv x-y}$. Aunque a priori no podamos decir exáctamente cúal será la cota inferior, parece claro que existirá una cota y eso será suficiente para verificar la terminación.

Nota: No tiene por qué haber una única expresión cota para cada bucle. En este caso también se podría haber elegido, por ejemplo, $\boldsymbol{E\equiv x-y^2}$ *ya que también decrecerá y será un número natural.*

Recordemos que el invariante era:

$$/*\, INV \,*/ \equiv \forall\, k\,(1\le k<y \rightarrow k*k<x)$$

Pasemos ahora a verificar las dos propiedades necesarias para asegurar la terminación del bucle:

$$\boldsymbol{INV \wedge B \rightarrow \exists c\,(E>c)}$$

1. $$\forall\, k\,(1\le k<y \rightarrow k*k<x) \wedge y*y<x \rightarrow x-y>0 \rightarrow \exists \boldsymbol{c}\,(\boldsymbol{E>c})$$

   $$/*\, \boldsymbol{INV \wedge B \wedge E=v} \,*/ \boldsymbol{I;}\ /*\, \boldsymbol{E<v} \,*/$$

2. $$\begin{array}{l} \forall\, k\,(1\le k<y \rightarrow k*k<x) \wedge y*y<x \wedge \boldsymbol{x-y=v} \\ \qquad\qquad \rightarrow x-(y+\boldsymbol{1}-\boldsymbol{1})=v \end{array}$$

3. $$\begin{array}{l} /*\, x-(y+1-1)=v \,*/\ \boldsymbol{y=y+1;} \\ /*\, x-(\boldsymbol{y-1})=v \,*/\,(\boldsymbol{AA}) \end{array}$$

4. $$\begin{array}{l} x-(y-1)=v \rightarrow x-y=v-1 \\ \qquad\qquad \rightarrow E<v \end{array}$$

5. $$/*\, \boldsymbol{INV \wedge B \wedge E=v} \,*/ \boldsymbol{I;}\ /*\, \boldsymbol{E<v} \,*/\,(\boldsymbol{RCN}\ \mathbf{2,3,4})$$

Ejemplo 2:

$$/*\ suma = 0 \wedge k = 1\ */ \equiv /*\ \Phi\ */$$
$$\textbf{while}(k \leq n)$$
$$\{$$
$$\textbf{suma} = \textbf{suma} + A[k];$$
$$k = k + 1;$$
$$\{$$
$$/*\ suma = \sum_{i=1}^{n} A[i]\ */ \equiv /*\ \Psi\ */$$

Recordemos que el invariante era:

$$/*\ INV\ */ \equiv /*\ suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1\ */$$

En este segundo ejemplo, *k* es la variable que crecerá, por lo que podemos elegir $E \equiv n - k$, ya que *k* irá creciendo hacia el último elemento del array (*n*), por lo que la expresión cota elegida decrecerá, y siempre será un número natural.

$$INV \wedge B \rightarrow \exists c\,(E > c)$$

1. $$suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \wedge k \leq n \rightarrow n - k > 0 \rightarrow \exists c\,(E > c)$$

   $$/*\ INV \wedge B \wedge E = v\ */\ \textbf{I};\ /*\ E < v\ */$$

2. $$suma = \sum_{i=1}^{k-1} A[i] \wedge 1 \leq k \leq n+1 \wedge k \leq n \wedge n - k = v$$
   $$\rightarrow n - k = v$$

3. $$/*\ n - k = v\ */\ \textbf{suma} = \textbf{suma} + A[k];\ /*\ n - k = v\ */ \equiv /*\ \Phi_1\ */\ (AA)$$

4. $$/*\ INV \wedge B \wedge E = v\ */$$
   $$\textbf{suma} = \textbf{suma} + A[k];\ /*\ \Phi_1\ */\ (RCN\ 2,3)$$

5. $$n - k = v \rightarrow n - (k + 1 - 1) = v$$

6. $$/*\ n - (k + 1 - 1) = v\ */\ k = k + 1;\ /*\ n - (k - 1) = v\ */\ (AA)$$

7. $$n - (k - 1) = v \rightarrow n - k = v - 1 \rightarrow E < v$$

8. $$/*\ \Phi_1\ */\ k = k + 1;\ /*\ E < v\ */\ (RCN\ 5,6,7)$$

9. $/* INV \wedge B \wedge E = v */$ **I ;** $/* E < v */$ $(\textbf{RCP 4,8})$

Ejercicios Verificación (IV)

Determina **las post-condiciones y los invariantes** correspondientes a los siguientes programas:

1.
```
/* i=1 */ ≡ /* Φ */
while (i≤n)
{
   A[i]=i;
   i=i+1;
}
/* Ψ */
```

2.
```
/* k>1 ∧ i=1 ∧ n=0 */ ≡ /* Φ */
while (i<k)
{
i=i+1;
if (i%3==0)
        n=n+1;
}
/* Ψ */
```

3.
```
/* i=1 ∧ num_neg=0 */ ≡ /* Φ */
while (i≤n)
{
if (A[i]<0)
  num_neg=num_neg+1;

i=i+1;
}
/* Ψ */
```

Verifica la **corrección total** de los siguientes programas:

4.
```
/* x=a ∧ y=b≥0 */ ≡ /* Φ */
while (y !=0)
{
        x=x+1;
        y=y−1;
}
/* x=a+b ∧ y=0 */ ≡ /* Ψ */
```

5.

```
/*i=1 ∧ j>1 ∧ sum=0 */ ≡ /* Φ */
while(j-i ≥ 0)
{
        sum = sum + i;
          i = i + 1;
}
/*sum = Σ_{k=1}^{i} k */ ≡ /* Ψ */
```

6.

```
/* f = 1 ∧ t = x > 1 */ ≡ /* Φ */
while(t ≥ 1)
{
        f = f * t;
        t = t - 1;
}
/* f = x! */ ≡ /* Ψ */
```

Soluciones propuestas Verificación (IV)

1.

```
/* i=1 */ ≡ /* Φ */
while(i≤n)
{
   A[i]=i;
   i=i+1;
}
/* Ψ */
```

Lo primero es determinar cuál es el objetivo o post-condición del programa. En este caso, el programa escribe en cada una de las posiciones del vector el índice *i* correspondiente. Así, pues la post-condición será:

$$/* \Psi */ \equiv /* \forall\ j(1 \le j \le n \rightarrow A[j] = j) */$$

Nótese que no se utiliza *i* como variable ligada del cuantificador. El objetivo es evitar posibles confusiones con la variable libre del programa.

Lo siguiente será determinar el invariante, que indicará qué propiedades se cumplen en todas las iteraciones:

$$/* INV */ \equiv /* \forall\ j(1 \le j \le i-1 \rightarrow A[j] = j) \wedge 1 \le i \le n+1 */$$

2.

```
/* k>1 ∧ i=1 ∧ n=0 */ ≡ /* Φ */
while(i<k)
{
i=i+1;
if(i%3==0)
        n=n+1;
}
/* Ψ */
```

$$/* \Psi */ \equiv /* n = \aleph\ j(2 \le j \le k \rightarrow j\ mod\ 3 = 0) */$$

$$/* INV */ \equiv /* n = \aleph\ j(2 \le j \le i \rightarrow j\ mod\ 3 = 0) \wedge 1 \le i \le k */$$

3.

```
/*i=0∧n≥1∧num_neg=0*/≡/*Φ*/
while(i<n)
{
i=i+1;
if(A[i]<0)
    num_neg=num_neg+1;
}
/*Ψ*/
```

$$/*\Psi*/ \equiv /*\text{num\_neg} = \aleph\ j (1 \le j \le n \rightarrow A[j] < 0)*/$$

$$/*\mathit{INV}*/ \equiv /*\text{num\_neg} = \aleph\ j (1 \le j \le i \rightarrow A[j] < 0) \wedge 0 \le i \le n*/$$

4.

```
/*x=a∧y=b≥0*/≡/*Φ*/
while(y!=0)
{
    x=x+1;≡A1
                /*Φ1*/
    y=y-1;≡A2
}
/*x=a+b∧y=0*/≡/*Ψ*/
```

Primero determinaremos el invariante. Será el siguiente:

$$/*\mathit{INV}*/ \equiv /*\ x + y = a + b \wedge 0 \le y \le b*/$$

Luego haremos la descomposición del programa:

$$(\boldsymbol{A})\ \Phi \rightarrow \mathit{INV}$$
$$(\boldsymbol{B})\ \mathit{INV} \rightarrow \text{def}\,(B)$$
$$(\boldsymbol{C})\ /*\ \mathit{INV} \wedge B*/\ A_1\ ;\ A_2\ ;\ /*\ \mathit{INV}*/$$
$$(\boldsymbol{C.1})\ /*\ \mathit{INV} \wedge B*/\ A_1\ ;\ /*\ \Phi_1*/$$
$$(\boldsymbol{C.2})\ /*\ \Phi_1*/\ A_2\ ;\ /*\ \mathit{INV}*/$$
$$(\boldsymbol{D})\ \mathit{INV} \wedge \neg B \rightarrow \Psi$$

Ahora ya podemos empezar el cálculo de Hoare para verificar la corrección parcial:

$(\boldsymbol{A})\ \boldsymbol{\Phi \rightarrow \mathit{INV}}$

1. $x = a \wedge y = b \ge 0 \rightarrow x + y = a + b \wedge 0 \le y \le b$

   $(\boldsymbol{B})\ \boldsymbol{\mathit{INV} \rightarrow \text{def}\,(B)}$

2. $x + y = a + b \wedge 0 \le y \le b \rightarrow \text{def}\,(y\,! = 0)$

   $(\boldsymbol{C.1})\ /*\ \boldsymbol{\mathit{INV} \wedge B}*/\ \boldsymbol{A_1\ ;\ /*\ \Phi_1}*/$

3. $x + y = a + b \wedge 0 \le y \le b \wedge y \ne 0 \rightarrow (x + 1) + (y - 1) = a + b \wedge 0 < y \le b$

4. $/*(x+1)+(y-1)=a+b \wedge 0<y\leq b*/\ \boldsymbol{x=x+1;}$
   $/*x+(y-1)=a+b \wedge 0<y\leq b*/ \equiv /*\boldsymbol{\Phi_1}*/(\boldsymbol{AA})$
5. $/*INV \wedge B*/\ \boldsymbol{A_1;}\ /*\Phi_1*/(\boldsymbol{RCN}\ \mathbf{3,4})$
   $(\boldsymbol{C.2})/*\boldsymbol{\Phi_1}*/\ \boldsymbol{A_2;}\ /*\boldsymbol{INV}*/$
6. $x+(y-1)=a+b \wedge 0<y\leq b \rightarrow x+(y-1)=a+b \wedge 0\leq y-1\leq b$
7. $/*x+(y-1)=a+b \wedge 0\leq y-1\leq b*/\ \boldsymbol{y=y-1;}$
   $/*x+y=a+b \wedge 0\leq y\leq b*/ \equiv /*\ \boldsymbol{INV}\ */(\boldsymbol{AA})$
8. $/*\Phi_1*/\ \boldsymbol{A_2;}\ /*INV*/(\boldsymbol{RCN}\ \mathbf{6,7})$
   $(\boldsymbol{C})/*\ \boldsymbol{INV} \wedge \boldsymbol{B}*/\ \boldsymbol{A_1;}\ \boldsymbol{A_2;}\ /*\ \boldsymbol{INV}*/$
9. $/*INV \wedge B*/\ A_1;\ A_2;\ /*INV*/(\boldsymbol{RCP}\ \mathbf{5,8})$
   $(\boldsymbol{D})\ \boldsymbol{INV} \wedge \neg \boldsymbol{B} \rightarrow \Psi$
10. $x+y=a+b \wedge 0\leq y\leq b \wedge y=0 \rightarrow x=a+b \wedge y=0$
11. $/*\Phi*/\ \boldsymbol{while}(\boldsymbol{B})\{\boldsymbol{A_1;}\ \boldsymbol{A_2;}\}/*\Psi*/(\boldsymbol{RWH}\ \mathbf{1,2,9,10})$

Ahora que hemos demostrado la corrección parcial del programa, verificaremos la terminación del mismo para poder llegar a verificar la corrección total.

Para ello, primero elegiremos una expresión cota. En este caso, la más simple será *y*, ya que la variable decrece hasta llegar a 0. Por lo tanto, ***E=y***.

$\boldsymbol{INV} \wedge \boldsymbol{B} \rightarrow \boldsymbol{E} \in \mathbb{N}$

1. $x+y=a+b \wedge \mathbf{0}\leq \boldsymbol{y} \leq \boldsymbol{b} \wedge \boldsymbol{y} \neq \mathbf{0} \rightarrow y \in \mathbb{N}$
   $/*\ \boldsymbol{INV} \wedge \boldsymbol{B} \wedge \boldsymbol{E}=\boldsymbol{v}*/\ \boldsymbol{I;}\ /*\ \boldsymbol{E}<\boldsymbol{v}*/$
2. $x+y=a+b \wedge 0\leq y\leq b \wedge y\neq 0 \wedge \boldsymbol{y}=\boldsymbol{v} \rightarrow y=v$
3. $/*y=v*/\ \boldsymbol{x=x+1;}\ /*\ y=v*/ \equiv /*\boldsymbol{\Phi_2}*/(\boldsymbol{AA})$
4. $/*\ INV \wedge B \wedge E=v*/\ A_1;\ /*\Phi_2*/(\boldsymbol{RCN}\ \mathbf{2,3})$
5. $y=v \rightarrow y-1=v-1$
6. $/*\ y-1=v-1*/\ \boldsymbol{y=y-1;}\ /*\ y=v-1*/(\boldsymbol{AA})$
7. $y=v-1 \rightarrow y<v$
8. $/*\Phi_2*/\ \boldsymbol{A_2;}\ /*E<v*/(\boldsymbol{RCN}\ \mathbf{5,6,7})$
9. $/*\ INV \wedge B \wedge E=v*/\ \boldsymbol{I;}\ /*\ E<v*/(\boldsymbol{RCP}\ \mathbf{4,8})$

5.

$/* i=1 \wedge j>1 \wedge \text{sum}=0 */ \equiv /* \Phi */$
`while``(j-i>=0)`
`{`
`sum=sum+i;`
`i=i+1;`
`}`
$/* \text{sum}=\sum_{k=1}^{j} k */ \equiv /* \Psi */$

Invariante:

$$/* INV */ \equiv /* \text{sum}=\sum_{k=1}^{i-1} k \wedge 1 \leq i \leq j+1 */$$

Descomposición:

$(\boldsymbol{A}) \Phi \rightarrow INV$
$(\boldsymbol{B}) INV \rightarrow \text{def}(B)$
$(\boldsymbol{C}) /* INV \wedge B */ A_1 ; A_2 ; /* INV */$
$(\boldsymbol{C.1}) /* INV \wedge B */ A_1 ; /* \Phi_1 */$
$(\boldsymbol{C.2}) /* \Phi_1 */ A_2 ; /* INV */$
$(\boldsymbol{D}) INV \wedge \neg B \rightarrow \Psi$

Cálculo de Hoare:

$(\boldsymbol{A}) \boldsymbol{\Phi} \rightarrow \boldsymbol{INV}$

1. $i=1 \wedge j>1 \wedge \text{sum}=0 \rightarrow \text{sum}=\sum_{k=1}^{i-1} k \wedge 1 \leq i \leq j+1$

$(\boldsymbol{B}) \boldsymbol{INV} \rightarrow \textbf{def}(\boldsymbol{B})$

2. $\text{sum}=\sum_{k=1}^{i-1} k \wedge 1 \leq i \leq j+1 \rightarrow \text{def}(j-i \geq 0)$

$(\boldsymbol{C.1}) /* \boldsymbol{INV \wedge B} */ \boldsymbol{A_1} ; /* \boldsymbol{\Phi_1} */$

3. $\text{sum}=\sum_{k=1}^{i-1} k \wedge 1 \leq i \leq j+1 \wedge j-i \geq 0$
$\rightarrow \text{sum}+i=\sum_{k=1}^{i} k \wedge 1 \leq i \leq j+1 \wedge j-i \geq 0$

4. $/* \text{sum}+i=\sum_{k=1}^{i} k \wedge 1 \leq i \leq j+1 \wedge j-i \geq 0 */$ **`sum=sum+i;`**
$/* \text{sum}=\sum_{k=1}^{i} k \wedge 1 \leq i \leq j+1 \wedge j-i \geq 0 */ \equiv /* \boldsymbol{\Phi_1} */ (\boldsymbol{AA})$

5. $/* INV \wedge B */ A_1 ; /* \Phi_1 */ (\textbf{RCN 3,4})$

$(\boldsymbol{C.2})$ /* $\boldsymbol{\Phi_1}$ */ $\boldsymbol{A_2}$; /* $\boldsymbol{INV}$ */

6. $\text{sum} = \sum_{k=1}^{i} k \wedge \mathbf{1} \le \boldsymbol{i} \le \boldsymbol{j}+\mathbf{1} \wedge \boldsymbol{j}-\boldsymbol{i} \ge \mathbf{0}$
   $\rightarrow \text{sum} = \sum_{k=1}^{i+1-1} k \wedge \mathbf{1} \le \boldsymbol{i}+\mathbf{1} \le \boldsymbol{j}+\mathbf{1}$

7. /*sum = $\sum_{k=1}^{i+1-1} k \wedge 1 \le i+1 \le j+1$ */ $\boldsymbol{i} = \boldsymbol{i}+\mathbf{1}$;
   /*sum = $\sum_{k=1}^{i-1} k \wedge 1 \le i \le j+1$ */ $\equiv$ /* $\boldsymbol{INV}$ */ $(\boldsymbol{AA})$

8. /* Φ_1 */ $\boldsymbol{A_2}$; /* INV */ $(\boldsymbol{RCN}\ \mathbf{6,7})$

$(\boldsymbol{C})$ /* $\boldsymbol{INV} \wedge \boldsymbol{B}$ */ $\boldsymbol{A_1}$; $\boldsymbol{A_2}$; /* $\boldsymbol{INV}$ */

9. /* $INV \wedge B$ */ A_1 ; A_2 ; /* INV */ $(\boldsymbol{RCP}\ \mathbf{5,8})$

$(\boldsymbol{D})\ \boldsymbol{INV} \wedge \neg \boldsymbol{B} \rightarrow \Psi$

$\text{sum} = \sum_{k=1}^{i-1} k \wedge 1 \le i \le j+1 \wedge j-i < 0$

10. $\rightarrow \text{sum} = \sum_{k=1}^{i-1} k \wedge \boldsymbol{i} = \boldsymbol{j}+\mathbf{1}$

$\rightarrow \text{sum} = \sum_{k=1}^{j} k \wedge 1 \le i \le j+1 \equiv$ /* Ψ */

11. /* Φ */ **while** $(\boldsymbol{B})$ { $\boldsymbol{A_1}$; $\boldsymbol{A_2}$; } /* Ψ */ $(\boldsymbol{RWH}\ \mathbf{1,2,9,10})$

Terminación:

Para ello, primero elegiremos una expresión cota. En este caso, la más simple será *j-i*, ya que *i* va creciendo hacia *j*. Por lo tanto, ***E=j-i***.

$\boldsymbol{INV} \wedge \boldsymbol{B} \rightarrow \exists \boldsymbol{c}(\boldsymbol{E} > \boldsymbol{c})$

1. $\text{sum} = \sum_{k=1}^{i-1} k \wedge 1 \le i \le j+1 \wedge \boldsymbol{j}-\boldsymbol{i} \ge \mathbf{0} \rightarrow E > -1$
   /* $\boldsymbol{INV} \wedge \boldsymbol{B} \wedge \boldsymbol{E} = \boldsymbol{v}$ */ $\boldsymbol{I}$; /* $\boldsymbol{E} < \boldsymbol{v}$ */
2. $\text{sum} = \sum_{k=1}^{i-1} k \wedge 1 \le i \le j+1 \wedge j-i \ge 0 \wedge j-i = v \rightarrow j-i = v$
3. /* $j-i=v$ */ **sum** = **sum** + **i** ;
   /* $j-i=v$ */ $\equiv$ /* $\boldsymbol{\Phi_2}$ */ $(\boldsymbol{AA})$
4. /* $INV \wedge B \wedge E = v$ */ A_1 ; /* Φ_2 */ $(\boldsymbol{RCN}\ \mathbf{2,3})$
5. $j-i=v \rightarrow j-(i+1-1)=v$
6. /* $j-(\boldsymbol{i}+\mathbf{1}-1)=v$ */ $\boldsymbol{i} = \boldsymbol{i}+\mathbf{1}$; /* $j-(\boldsymbol{i}-1)=v$ */ $(\boldsymbol{AA})$
7. $j-(i-1)=v \rightarrow \boldsymbol{j}-\boldsymbol{i} = \boldsymbol{v}-\mathbf{1} < \boldsymbol{v} \equiv$ /* $\boldsymbol{E} < \boldsymbol{v}$ */
8. /* Φ_2 */ $\boldsymbol{A_2}$; /* $E < v$ */ $(\boldsymbol{RCN}\ \mathbf{5,6,7})$
9. /* $INV \wedge B \wedge E = v$ */ $\boldsymbol{I}$; /* $E < v$ */ $(\boldsymbol{RCP}\ \mathbf{4,8})$

6.

```
/* f = 1 ∧ t = x > 1 */ ≡ /* Φ */
while (t ≥ 1)
{
        f = f * t;
        t = t − 1;
}
/* f = x! */ ≡ /* Ψ */
```

Invariante:

$$/* INV */ \equiv /* \ f = \prod_{i=t+1}^{x} k \wedge 0 \leq t \leq x */$$

Descomposición:

$$(\boldsymbol{A}) \Phi \rightarrow INV$$
$$(\boldsymbol{B}) \ INV \rightarrow \text{def}\ (B)$$
$$(\boldsymbol{C}) /* \ INV \wedge B */ A_1 ; A_2 ; /* \ INV */$$
$$(\boldsymbol{C.1}) /* \ INV \wedge B */ A_1 ; /* \Phi_1 */$$
$$(\boldsymbol{C.2}) /* \Phi_1 */ A_2 ; /* \ INV */$$
$$(\boldsymbol{D}) \ INV \wedge \neg B \rightarrow \Psi$$

Cálculo de Hoare:

$(\boldsymbol{A}) \Phi \rightarrow \boldsymbol{INV}$

1. $f = 1 \wedge t = x \leq 1 \rightarrow f = \prod_{i=t+1}^{x} i \wedge 0 \leq t \leq x$

$(\boldsymbol{B})\ \boldsymbol{INV} \rightarrow \textbf{def}\ (\boldsymbol{B})$

2. $f = \prod_{i=t+1}^{x} i \wedge 0 \leq t \leq x \rightarrow \text{def}\ (t \geq 1)$

$(\boldsymbol{C.1}) /* \ \boldsymbol{INV} \wedge \boldsymbol{B} */ \boldsymbol{A_1} ; /* \boldsymbol{\Phi_1} */$

3. $f = \prod_{i=t+1}^{x} i \wedge 0 \leq t \leq x \wedge t \geq 1 \rightarrow f * t = \prod_{i=t}^{x} i \wedge 1 \leq t \leq x$

4. $/* \ f * t = \prod_{i=t}^{x} i \wedge 1 \leq t \leq x */ \ \boldsymbol{f = f * t;}$

5. $/* \ f = \prod_{i=t}^{x} i \wedge 1 \leq t \leq x */ \equiv /* \boldsymbol{\Phi_1} */ (\boldsymbol{AA})$

$/* \ INV \wedge B */ A_1 ; /* \Phi_1 */ (\boldsymbol{RCN\ 3,4})$

$(\boldsymbol{C.2}) /* \boldsymbol{\Phi_1} */ \boldsymbol{A_2} ; /* \ \boldsymbol{INV} */$

6. $f=\prod_{i=t}^{x} i \wedge 0 \le t \le x$
   $\rightarrow f=\prod_{i=t-1+1}^{x} i \wedge 0 \le t-1 \le x$
7. $/* f=\prod_{i=t-1+1}^{x} i \wedge 0 \le t-1 \le x */ \mathbf{t=t-1;}$
   $/* f=\prod_{i=t+1}^{x} i \wedge 0 \le t \le x */ \equiv /* \mathbf{INV} */ (\mathbf{AA})$
8. $/* \Phi_1 */ \mathbf{A_2;}\ /* INV */ (\mathbf{RCN}\ 6,7)$
   $(\mathbf{C}) /* \mathbf{INV} \wedge \mathbf{B} */ \mathbf{A_1;}\ \mathbf{A_2;}\ /* \mathbf{INV} */$
9. $/* INV \wedge B */ A_1; A_2;\ /* INV */ (\mathbf{RCP}\ 5,8)$
   $(\mathbf{D})\ \mathbf{INV} \wedge \neg \mathbf{B} \rightarrow \Psi$
10. $f=\prod_{i=t+1}^{x} i \wedge 0 \le t \le x \wedge t<1$
    $\rightarrow f=\prod_{i=t+1}^{x} i \wedge t=0$
    $\rightarrow f=x! \equiv /* \Psi */$
11. $/* \Phi */ \mathbf{while}(\mathbf{B})\{\mathbf{A_1;}\ \mathbf{A_2;}\}\ /* \Psi */ (\mathbf{RWH}\ 1,2,9,10)$

Terminación:

En este caso, la expresión cota será *t*, ya que decrecerá hasta llegar a valer 0. Por lo tanto, ***E=t***.

$\mathbf{INV} \wedge \mathbf{B} \rightarrow \exists \mathbf{c}(\mathbf{E}>\mathbf{c})$

1. $f=\prod_{i=t+1}^{x} k \wedge 0 \le t \le x \wedge \mathbf{t} \ge \mathbf{1} \rightarrow E>0$
   $/* \mathbf{INV} \wedge \mathbf{B} \wedge \mathbf{E}=\mathbf{v} */ \mathbf{I;}\ /* \mathbf{E}<\mathbf{v} */$
2. $f=\prod_{i=t+1}^{x} k \wedge 0 \le t \le x \wedge t \ge 1 \wedge t=v \rightarrow t=v$
3. $/* t=v */ \mathbf{f=f*t;}\ /* t=v */ \equiv /* \Phi_2 */ (\mathbf{AA})$
4. $/* INV \wedge B \wedge E=v */ A_1;\ /* \Phi_2 */ (\mathbf{RCN}\ 2,3)$
5. $t=v \rightarrow t-1+1=v$
6. $/* t-1+1=v */ \mathbf{t=t-1;}\ /* t+1=v */ (\mathbf{AA})$
7. $t+1=v \rightarrow \mathbf{t=v-1<v} \equiv /* \mathbf{E}<\mathbf{v} */$
8. $/* \Phi_2 */ \mathbf{A_2;}\ /* E<v */ (\mathbf{RCN}\ 5,6,7)$
9. $/* INV \wedge B \wedge E=v */ \mathbf{I;}\ /* E<v */ (\mathbf{RCP}\ 4,8)$

3.10 Llamadas a funciones no recursivas

En una llamada a un subprograma (sólo estudiaremos las funciones) distinguiremos dos partes: **el programa que llama y el subprograma llamado**.

El subprograma llamado deberá ser verificado respecto a una especificación pre-post como se ha visto hasta el momento. El programa que llama a la función, en cambio, deberá asegurar que se cumple la precondición del subprograma, para que el resultado devuelto respete la postcondición del subprograma.

Para explicar cómo verificar una llamada a un subprograma primero veremos un ejemplo, donde el subprograma llamado será el siguiente:

```
sumaVec(int [] Vec, int n, int i, int j) return int suma
/*1≤i≤j≤n*/ ≡ /*Φ_sub*/
suma=0;
while(i≤j)
{
 suma=suma+Vec[i];
 i=i+1;
}
/*suma=∑_{k=i}^{j} Vec[k]*/ ≡ /*Ψ_sub*/
```

El programa principal:

```
/*2≤n*/ ≡ /*Φ*/
suma1=sumaVec(A, n, 1, n/2);
suma2=sumaVec(A, n, (n/2)+1, n);
/* suma1+suma2=∑_{k=1}^{n} A[k]*/ ≡ /*Ψ*/
```

En este ejemplo, primero debemos verificar el subprograma para asegurarnos de que cumple su especificación. Verificaremos este programa respecto a su precondición y

postcondición como hemos hecho hasta ahora. Una vez que lo verifiquemos, trataremos de verificar el programa llamador de la siguiente manera:

Antes de poder aplicar la especificación de la llamada, deberemos asegurarnos de que antes de hacerla siempre se cumple la precondición del subprograma para los parámetros que se le pasan a la función. La precondición es $/*1 \le i \le j \le n*/$, donde deberemos sustituir los parámetros por las expresiones pasadas como parámetros por el programa llamador: $/*1 \le 1 \le n/2 \le n*/$. Esta expresión tiene que poderse deducir de la aserción anterior a la llamada:

1. $2 \le n \rightarrow 2 \le n \wedge 1 \le 1 \le n/2 \le n$

Ahora ya sabemos que se cumplirá la precondición antes de la llamada, por lo que deberemos aplicar la especificación del subprograma

2. $/*2 \le n \wedge \mathbf{1} \le \mathbf{1} \le \boldsymbol{n/2} \le \boldsymbol{n}*/ \; suma1 = sumaVec(A, n, 1, n/2);$
   $/*2 \le n \wedge \boldsymbol{suma1} = \sum_{\boldsymbol{k}=\mathbf{1}}^{\boldsymbol{n/2}} \boldsymbol{A}[\boldsymbol{k}]*/ (\textbf{\textit{por llamada a subprogr.}})$
3. $/*2 \le n*/ \; suma1 = sumaVec(A, n, 1, n/2);$
   $/*2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k]*/ (\textbf{\textit{RCN 1,2}})$

Y haremos lo mismo para la segunda llamada:

4. $2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k]$
   $\rightarrow 2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k] \wedge 1 \le (n/2)+1 \le n \le n$
   $/*2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k] \wedge \mathbf{1} \le (\boldsymbol{n/2})+\mathbf{1} \le \boldsymbol{n} \le \boldsymbol{n}*/$
5. $suma2 = sumaVec(A, n, (n/2)+1, n);$
   $/*2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k] \wedge \boldsymbol{suma2} = \sum_{\boldsymbol{k}=(\boldsymbol{n/2})+\mathbf{1}}^{\boldsymbol{n}} \boldsymbol{A}[\boldsymbol{k}]*/ (\textbf{\textit{por llamada a subprogr.}})$
6. $2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k] \wedge suma2 = \sum_{k=(n/2)+1}^{n} A[k]$
   $\rightarrow suma1 + suma2 = \sum_{k=1}^{n} A[k] \equiv /*\Psi*/$
7. $/*2 \le n \wedge suma1 = \sum_{k=1}^{n/2} A[k]*/ \; suma2 = sumaVec(A, n, (n/2)+1, n);$
   $/* suma1 + suma2 = \sum_{k=1}^{n} A[k]*/ (\textbf{\textit{RCN 4,5,6}})$

8. $/*\Phi*/\ P\ ;\ /*\Psi*/\ (\mathbf{RCP\ 3,7})$

Lo que haremos, entonces, es deducir de la aserción a una llamada a subprograma la expresión resultante de sustituir en la precondición del subprograma (Φ_{sub}) los parámetros por los valores pasados. Después, aplicaremos la especificación del mismo, sustituyendo la precondición por la postcondición (Ψ_{sub}) después de la llamada:

$$/*\ \Omega \wedge \Phi_{sub}\ */\ var1 = funcion(p1, p2, \ldots, pn)\ /*\ \Omega \wedge \Psi_{sub}\ */$$

Donde Ω será la parte de la aserción anterior a la llamada independiente de los parámetros pasados a la función y la variable donde recogeremos el resultado devuelto.

3.11 Llamadas a funciones recursivas

Un subprograma recursivo es aquel que se llama a sí mismo. Los subprogramas recursivos suelen constar de uno o varios **casos simples y casos inductivos**.

Los *casos simples* (en la mayoría de los ejemplos con los que trabajaremos existirá uno únicamente) se resuelven directamente sin necesidad de llamadas recursivas. También se conocen como casos triviales porque la solución es muy fácil de calcular debido a la simpleza del problema a resolver.

Los *casos inductivos*, en cambio, requieren una o más llamadas recursivas para poder resolverlos. La complejidad del problema a resolver será menos para las llamadas recursivas y la solución se calculará en función de los resultados devueltos por las llamadas recursivas.

En el siguiente ejemplo:

```
factorial(int n)return int resul
{ /* n≥0 */
if (n==0) resul=1;
else
 {
  resul=factorial(n−1);
  resul=resul∗n;
 }
} /* resul=n! */
```

Se pretende calcular el factorial de *n*. Para ello se distinguen dos partes:

- **El caso simple:** Si *n* es igual a 0, entonces el factorial será 1, por lo que se hace *resul=1*.

- **El caso inductivo:** En cualquier otro caso (como sabemos por la precondición que *n>=0*, siempre se cumplirá *n>0* en el caso inductivo), primero se llama a *factorial(n-1)*. Suponiendo que la llamada recursiva cumpla su especificación, nos devolverá *(n-1)!*, por lo que le multiplicaremos n al resultado devuelto por la llamada para calcular *n!*.

Nótese que todas las instancias de la función *factorial()* se quedarán esperando a que la llamada recursiva devuelva el resultado, lo que no ocurrirá hasta que se llegue al caso simple (*n=0*), a partir de lo cual se irán multiplicando el resultado por el valor de *n* correspondiente a cada instancia de la función.

resul= 1(caso simple)*1*2*3*4...*(n-1)*n

3.11.1 Hipótesis de inducción

La *Hipótesis de Inducción* es el supuesto sobre el que se construye un programa recursivo, el comportamiento esperado de una llamada recursiva a partir de la especificación del programa desde que se hace.

En el caso anterior, la H*ipótesis de Inducción* será la siguiente:

$$/*n-1\geq 0*/\ resul=factorial(n-1);\ /*resul=(n-1)!*/$$

Se obtendrá sustituyendo los parámetros de la función recursiva con las expresiones pasadas a la llamada. En el caso anterior la especificación de la función factorial era:

$$/*n\geq 0*/\ P;\ /*resul=n!*/$$

Sabiendo que la llamada recursiva es:

$$resul=factorial(n-1);$$

Tendremos que sustituir *n* por *n-1* en la especifiación, y teniendo en cuenta que el resultado se guarda en resul, obtendremos la Hipótesis de Inducción:

$$/*n-1\geq 0*/\ resul=factorial(n-1);\ /*resul=(n-1)!*/$$

Sobre la hipótesis de que la llamada recursiva va a cumplir su especificación se construye la función recursiva, y nos hará falta enunciarla para poder verificar cualquier programa recursivo. **Una vez que la enunciemos la consideraremos verificada y la aplicaremos como si fuese una regla de inferencia más.**

3.11.2 Corrección parcial

Para verificar parcialmente un subprograma recursivo tendremos que verificar que tanto en el caso simple como en el inductivo se satisface la postcondición.

a) $/*\Phi \wedge B*/\ P_s\ ;\ /*\Psi*/$

b) $/*\Phi \wedge \neg B*/\ P_i\ ;\ /*\Psi*/$

Donde la parte del programa correspondiente al caso simple es P_s y la correspondiente al inductivo como P_i . Si hubiese más de un caso simple, habrá que hacer lo mismo para cada uno de ellos.

La verificación del ejemplo anterior sería la siguiente:

a) Caso simple: $/*\Phi \wedge B*/\ P_s\ ;\ /*\Psi*/$

1. $n \geq 0 \wedge n = 0 \rightarrow n = 0 \rightarrow n! = 1$
2. $/*n! = 1*/\ \texttt{resul} = 1\ ;\ /*resul = n!*/\ (\textbf{\textit{AA}})$
3. $/*\Phi \wedge B*/\ P_s\ ;\ /*\Psi*/\ (\textbf{\textit{RCN}}\ \textbf{1,2})$

b) Caso inductivo: $/*\Phi \wedge \neg B*/\ P_i\ ;\ /*\Psi*/$

1. $n \geq 0 \wedge n \neq 0 \rightarrow n > 0 \rightarrow n - 1 \geq 0$
2. $/*n-1 \geq 0*/\ \texttt{resul} = \texttt{factorial}(n-1)\ ;\ /*resul = (n-1)!*/\ (\textbf{\textit{Hip. Ind.}})$
3. $/*\Phi \wedge \neg B*/\ \texttt{resul} = \texttt{factorial}(n-1)\ ;\ /*resul = (n-1)!*/\ (\textbf{\textit{RCN}}\ \textbf{1,2})$
4. $resul = (n-1)! \rightarrow resul * n = n!$
5. $/*resul * n = n!*/\ \texttt{resul} = \texttt{resul} * n\ ;\ /*resul = n!*/\ (\textbf{\textit{AA}})$
6. $/*resul = (n-1)!*/\ \texttt{resul} = \texttt{resul} * n\ ;\ /*resul = n!*/\ (\textbf{\textit{RCN}}\ \textbf{4,5})$
7. $/*\Phi \wedge \neg B*/\ P_i\ ;\ /*\Psi*/\ (\textbf{\textit{RCP}}\ \textbf{3,6})$

3.11.3 Validación

Al igual que con las estructuras repetitivas, tendremos que verificar que la recursividad finalizará en un **número finito de pasos** para poder verificar la corrección total de un programa recursivo.

El primer paso será, como con los **while**, elegir la expresión cota *E*. Después deberemos verificar que exista una cota inferior para *E* y para la expresión cota para la llamada recurisva (*E'*), y que *E'* sea menor que *E* (el valor de la expresión tiene que decrecer de una llamada a otra).

La expresión **E deberá estar formada por variables y operaciones enteras**, exactamente igual que verificando bucles.

Así pues, tendremos que verificar:

a) Caso simple: $\exists c(E > c)$
b) Caso inductivo: $\exists c(E > c \wedge E' > c) \wedge E > E'$

En el ejemplo del factorial, la expresión cota más sencilla será $E \equiv n$ ya que decrecerá hasta llegar al caso simple y siempre será mayor que cero:

a) Caso simple: $\Phi \wedge B \equiv n \geq 0 \wedge n = 0 \rightarrow n > 0 \rightarrow \exists c(E > c)$
b) Caso inductivo: $E' \equiv n - 1$

$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n > 0 \rightarrow n > -1 \wedge n - 1 \geq -1 \rightarrow \exists c(E > c \wedge E' > c)$$

$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n > 0 \rightarrow n > n - 1 \rightarrow E > E'$$

Después de esto, el programa recursivo se considerará **totalmente correcto**.

Ejercicios Verificación (V)

Verificar la **corrección total** de los siguientes **programas principales**

1.

Programa principal:

/* *true* */
`maxXY = max(x, y);`
`maxXZ = max(x, z);`
`maximo = max(maxXY, maxXZ);`
/* $(maximo = x \vee maximo = y \vee maximo = z) \wedge (maximo \geq x \wedge maximo \geq y \wedge maximo \geq z)$ */

Especificación función llamada:

$max(\text{int } a, \text{int } b)\ return\ \text{int maxAB}$
/* *true* */
/* $(maxAB = a \vee maxAB = b) \wedge (maxAB \geq a \wedge maxAB \geq b)$ */

2.

Programa principal:

/* $y + 5 \geq 5$ */
`z = Potencia(x, y-1);`
`z = z*x;`
/* $z = x^y$ */

Especificación función llamada:

$Potencia(\text{int } a, \text{int } b)\ return\ \text{int}\ pot$
/* $b \geq 0$ */
/* $pot = a^b$ */

Verificar la **corrección total** de los siguientes **programas recursivos**:

3.

```
sumSerie(int n) return int sumS
/* 0≤n */
if(n==0)
 sumS=0;
else
{
 sumS=sumSerie(n-1);
 sumS=sumS+n*n;
 sumS=sumS+n;
}
/* sum= Σ_{i=1}^{n} i^2+i */
```

4.

```
mediaVector(int [] A, int i) return int media
/* 1≤i≤n */
if(i==n)
 media=A[n]/n;
else
{
 media=mediaVector(A,i+1);
 media=media+(A[i]/n);
}
/* media= (Σ_{k=i}^{n} A[k]) / n */
```

Soluciones propuestas Verificación (V)

1.

Programa principal:

/* *true* */
`m1=max(x,y);`
`m2=max(m1,z);`
/* $(m2=x \vee m2=y \vee m2=z) \wedge (m2 \geq x \wedge m2 \geq y \wedge m2 \geq z)$ */

Especificación función llamada:

$max(\text{int } a, \text{int } b)$ *return* int maxAB
/* *true* */
/* $(maxAB=a \vee maxAB=b) \wedge (maxAB \geq a \wedge maxAB \geq b)$ */

1. /* ***true*** */ `m1=max(x,y);` /* $(\mathbf{m1}=\mathbf{x} \vee \mathbf{m1}=\mathbf{y}) \wedge (\mathbf{m1} \geq \mathbf{x} \wedge \mathbf{m1} \geq \mathbf{y})$ */
   (***llamada a subprogr.***)

2. $(m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y)$
   $\rightarrow (m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y) \wedge \mathbf{true}$

3. /* $(m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y) \wedge \mathbf{true}$ */
   `m2=max(m1,z);`
   /* $(m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y) \wedge (\mathbf{m2}=\mathbf{m1} \vee \mathbf{m2}=\mathbf{z}) \wedge (\mathbf{m2} \geq \mathbf{m1} \wedge \mathbf{m2} \geq \mathbf{z})$ */
   (***llamada a subprogr.***)

4. $(m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y) \wedge (m2=m1 \vee m2=z) \wedge (m2 \geq m1 \wedge m2 \geq z)$
   $\rightarrow (m2=x \vee m2=y \vee m2=z) \wedge (m2 \geq x \wedge m2 \geq y \wedge m2 \geq z)$

5. /* $(m1=x \vee m1=y) \wedge (m1 \geq x \wedge m1 \geq y)$ */ `m2=max(m1,z);`
   /* $(m2=x \vee m2=y \vee m2=z) \wedge (m2 \geq x \wedge m2 \geq y \wedge m2 \geq z)$ */ (***RCN* 2,3,4**)

6. /* *true* */
   `m1=max(x,y);`
   `m2=max(m1,z);`
   /* $(m2=x \vee m2=y \vee m2=z) \wedge (m2 \geq x \wedge m2 \geq y \wedge m2 \geq z)$ */ (***RCP* 1,5**)

2.

Programa principal:

/* $y+5\geq 5$ */
`z=Potencia(x,y-1);`
`z=z*x;`
/* $z=x^y$ */

Especificación función llamada:

Potencia(int *a*, int *b*) *return* int *pot*
/* $b\geq 0$ */
/* $pot=a^b$ */

1. $y+5\geq 5 \rightarrow y\geq 0$
2. /* $y\geq 0$ */ `z=Potencia(x,y-1);` /* $z=x^{(y-1)}$ */ (***por llamada a subprogr.***)
3. /* $y+5\geq 5$ */ `z=Potencia(x,y-1);` /* $z=x^{(y-1)}$ */ (***RCN* 1,2**)
4. $z=x^{(y-1)} \rightarrow z*x=x^y$
5. /* $z*x=x^y$ */ $z=z*x$; /* $z=x^y$ */ (***AA***)
6. /* $z=x^{(y-1)}$ */ $z=z*x$; /* $z=x^y$ */ (***RCN* 4,5**)
7. /* $y+5\geq 5$ */ `z=Potencia(x,y-1);` `z=z*x;` /* $z=x^y$ */ (***RCP* 3,6**)

3.

```
sumSerie(int n) return int sumS
/*0≤n*/
if(n==0)
 sumS=0;
else
{
 sumS=sumSerie(n-1);
 sumS=sumS+n*n;
 sumS=sumS+n;
}
/*sum=Σ_{i=1}^{n} i^2+i*/
```

Caso simple: $/*\Phi \wedge B*/\ P_{simple}\ ;\ /*\Psi*/$

1. $\Phi \wedge B \equiv 0 \leq n \wedge n = 0 \rightarrow \sum_{i=1}^{n} \boldsymbol{i^2 + i = 0}$

2. $/*\sum_{i=1}^{n} i^2 + i = 0*/\ sumS = 0\ ;/*\ sumS = \sum_{i=1}^{n} i^2 + i*/ (\boldsymbol{AA})$

3. $/*0 \leq n \wedge n = 0*/\ sumS = 0\ ;/*\ sumS = \sum_{i=1}^{n} i^2 + i*/ (\boldsymbol{RCN\ 1,2})$

Caso inductivo: $/*\Phi \wedge \neg B*/\ P_{inductivo}\ ;\ /*\Psi*/$

Hipótesis de Inducción $\equiv /*\ \boldsymbol{0 \leq n-1}\ */\ \boldsymbol{sumS = sumSerie(n-1)\ ;\ /*\ sumS = \sum_{i=1}^{n-1} i^2 + i\ */}$

4. $\Phi \wedge \neg B \equiv 0 \leq n \wedge n \neq 0 \rightarrow 0 < n \rightarrow 0 \leq n-1$

5. $/*0 \leq n-1*/\ sumS = sumSerie(n-1)\ ;/*\ sumS = \sum_{i=1}^{n-1} i^2 + i*/ (\boldsymbol{Hip.Ind.})$

6. $/*0 \leq n \wedge n \neq 0*/\ sumS = sumSerie(n-1)\ ;/*\ sumS = \sum_{i=1}^{n-1} i^2 + i*/ (\boldsymbol{RCN\ 4,5})$

7. $sumS = \sum_{i=1}^{n-1} i^2 + i \rightarrow sumS + n*n = (\sum_{i=1}^{n-1} i^2 + i) + n*n$

8. $/* sumS + n*n = (\sum_{i=1}^{n-1} i^2 + i) + n*n */ \; sumS = sumS + n*n;$
   $/* sumS = (\sum_{i=1}^{n-1} i^2 + i) + n*n */ (\boldsymbol{AA})$

9. $/* sumS = \sum_{i=1}^{n-1} i^2 + i */ \; sumS = sumS + n*n;$
   $/* sumS = (\sum_{i=1}^{n-1} i^2 + i) + n*n */ (\boldsymbol{RCN}\ \mathbf{7,8})$

10. $sumS = (\sum_{i=1}^{n-1} i^2 + i) + n*n \rightarrow sumS + n = (\sum_{i=1}^{n-1} i^2 + i) + (n*n + n)$
    $\rightarrow sumS + n = (\sum_{i=1}^{n} i^2 + i)$

11. $/* sumS + n = \sum_{i=1}^{n} i^2 + i */ \; sumS = sumS + n;$
    $/* sumS = \sum_{i=1}^{n} i^2 + i */ (\boldsymbol{AA})$

12. $/* sumS = (\sum_{i=1}^{n-1} i^2 + i) + n*n */ \; sumS = sumS + n;$
    $/* sumS = \sum_{i=1}^{n} i^2 + i */ (\boldsymbol{RCN}\ \mathbf{10,11})$

13. $/* \Phi \wedge \neg B */ P_{inductivo}; \; /* \Psi */ (\boldsymbol{RCP}\ \mathbf{6,9,12})$

Terminación: $E \equiv n$

a) Caso simple: $\Phi \wedge B \equiv n \geq 0 \wedge n = 0 \rightarrow n = 0 \rightarrow E > 0$

b) Caso inductivo: $E' \equiv n - 1$

$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n \neq 0 \rightarrow E > 0 \wedge E' > -1$$
$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n \neq 0 \rightarrow n > n - 1 \rightarrow E > E'$$

4.

```
mediaVector(int []A, int i) return int media
/*1≤i≤n*/
if(i==n)
 media=A[n]/n;
else
{
 media=mediaVector(A,i+1);
 media=media+(A[i]/n);
}
/*media=(sum_{k=i}^{n} A[k])/n*/
```

Caso simple: $/*\Phi \wedge B*/\ P_{simple}\ ;\ /*\Psi*/$

1. $\Phi \wedge B \equiv 1 \le i \le n \wedge \boldsymbol{i = n} \rightarrow \frac{(\sum_{k=i}^{n} A[k])}{n} = A[n]/n$

2. $/*\frac{(\sum_{k=i}^{n} A[k])}{n} = A[n]/n*/\ \boldsymbol{media = A[n]/n;}$

   $/*\frac{(\sum_{k=i}^{n} A[k])}{n} = media*/(\boldsymbol{AA})$

   $/*1 \le i \le n \wedge i = n*/\ \boldsymbol{media = A[n]/n;}$

3. $/*media = \frac{(\sum_{k=i}^{n} A[k])}{n}*/(\boldsymbol{RCN\ 1,2})$

Caso inductivo: $/*\Phi \wedge \neg B*/\ P_{inductivo}\ ;\ /*\Psi*/$

Hipótesis de Inducción $\equiv /*\mathbf{1 \le i+1 \le n*/}\ \boldsymbol{media = mediaVector(A, i+1);}$

$$/*\boldsymbol{media} = \frac{(\sum_{k=i+1}^{n} \boldsymbol{A[k]})}{\boldsymbol{n}}*/$$

4. $\Phi \wedge \neg B \equiv 1 \le i \le n \wedge i \ne n \rightarrow 1 \le i < n \rightarrow 1 \le i+1 \le n$

/*$1 \le i+1 \le n$*/ `media=mediaVector(A,i+1);`

5. /*$media = \frac{(\sum_{k=i+1}^{n} A[k])}{n}$*/ (***Hip.Ind.***)

/*$\Phi \wedge \neg B$*/ `media=mediaVector(A,i+1);`

6. /*$media = \frac{(\sum_{k=i+1}^{n} A[k])}{n}$*/ (***RCN* 4,5**)

7. $media = \frac{(\sum_{k=i+1}^{n} A[k])}{n} \rightarrow media + A[i]/n = \frac{(\sum_{k=i}^{n} A[k])}{n}$

8. /*$media + A[i]/n = \frac{(\sum_{k=i}^{n} A[k])}{n}$*/ `media=media+(A[i]/n);` /*
$media = \frac{(\sum_{k=i}^{n} A[k])}{n}$*/ (***AA***)

9. /*$media = \frac{(\sum_{k=i+1}^{n} A[k])}{n}$*/ `media=media+(A[i]/n);` /*
$media = \frac{(\sum_{k=i}^{n} A[k])}{n}$*/ (***RCN* 7,8**)

10. /*$\Phi \wedge B$*/ $P_{inductivo}$ */ Ψ */ (***RCP* 6,9**)

Terminación: $E \equiv n-i$

a) Caso simple: $\Phi \wedge B \equiv 1 \le i \le n \wedge i = n \rightarrow n-i = 0 \rightarrow E > -1$
b) Caso inductivo: $E' \equiv n-(i+1)$
$\Phi \wedge \neg B \equiv 1 \le i \le n \wedge i \ne n \rightarrow 1 \le i < n \rightarrow n-i > 0 \rightarrow E > -1 \wedge E' > -2$
$\Phi \wedge \neg B \equiv 1 \le i \le n \wedge i \ne n \rightarrow 1 \le i < n \rightarrow n-i > n-(i+1) \rightarrow E > E'$

4. DERIVACIÓN

El proceso de la programación conlleva inherentemente errores asociados, dado que la mayoría de las veces se hace intuitivamente y no de una manera razonada. Por ello, cuanto más automático sea el proceso de la programación, menor será la probabilidad de cometer errores.

La **derivación** nos ayudará a deducir cómo se deberá diseñar un programa a partir de una especificación pre-post de una manera razonada, permitiéndonos además obtener el programa perfectamente documentado.

Constará de cuatro pasos:

i) Especificación Pre-Post:
Primero especificaremos formalmente cuál es el objetivo del programa y sus requisitos. La corrección del programa será respecto a esta especificación.

ii) Análisis de casos:
Después analizaremos los posibles casos simples e inductivos y deduciremos formalmente cuál será el código, que se irá verificando parcialmente mientras se deduce.

iii) Validación de la inducción:
En este paso comprobaremos que la recursividad acabe en un número finito de pasos.

iv) Programa documentado:
Finalmente, obtendremos el programa entero documentado.

4.1 Derivación de funciones recursivas

Ilustraremos con un **ejemplo** cómo derivar programas recursivos. Trataremos de escribir un **programa recursivo que calcule el productorio de los n primeros números naturales**.

i) **Especificación Pre-Post:**

En este paso escribiremos la cabecera de la función recursiva junto a su especificación Pre-Post.

$$ProductorioN(\text{int } n)\ return \text{int } resul$$
$$PRE \equiv /* n \geq 0 */$$
$$POST \equiv /* resul = \prod_{i=1}^{n} i */$$

ii) **Análisis de casos:**

Primero, estudiaremos los posibles casos simples. Recordemos que los casos simples son aquellos que son tan triviales que se pueden resolver sin utilizar la recursividad.

a) Caso simple:

Como el objetivo es resolver un productorio desde 1 hasta *n*, el caso simple más obvio será cuando se cumpla *n<=1*. Cualquier valor de n que cumpla esa condición hará que el productorio sea 1 (por dominio vacío o porque sólo se multiplica el 1).

1. $n \leq 1 \rightarrow \prod_{i=1}^{n} i = 1$
2. $/* \prod_{i=1}^{n} i = 1 */$ **`resul=1;`** $/* \prod_{i=1}^{n} i = resul */ (\boldsymbol{AA})$
3. $/* \Phi \wedge B */$ **`resul=1;`** $/* \Psi */ (\boldsymbol{RCN}\ \mathbf{1,2})$

Por lo tanto, en el caso simple (cuando se cumpla *n<=1*) deberemos devolver 1 para satisfacer la postcondición.

b) Caso inductivo:

La idea de la recursión será ir decrementando el valor de *n* que se le pase a la llamada recursiva hasta llegar al caso trivial. A partir de la especificación del 1er paso podemos deducir la **Hipótesis de Inducción**:

$$/*\, n-1 \geq 0 \,*/\ \texttt{resul=ProductorioN}(n-1);\ /*\, resul = \prod_{i=1}^{n-1} i \,*/$$

Una vez tengamos la Hipótesis de Inducción podemos comenzar a derivar el caso inductivo. Partiremos de las propiedades que sabremos se cumplirán en el caso inductivo ($\Phi \wedge \neg B$) e intentaremos deducir que se cumple la precondición de la H.I. (en este caso true):

1. $n \geq 0 \wedge n > 1 \rightarrow n-1 \geq 0$

Ahora podremos aplicar la H.I. La llamada recursiva irá generalmente la primera en el caso inductivo para facilitar nuestro trabajo.

2. $/*\, n-1 \geq 0 \,*/\ \texttt{resul=ProductorioN}(n-1);\ /*\, resul = \prod_{i=1}^{n-1} i \,*/ (\textbf{\textit{H.I.}})$

3. $/*\, n \geq 0 \wedge n > 1 \,*/\ \texttt{resul=ProductorioN}(n-1);$
   $/*\, resul = \prod_{i=1}^{n-1} i \,*/ (\textbf{\textit{RCN}}\ \textbf{1,2})$

Sabiendo que tendremos en la variable *resul* el productorio desde 1 hasta n-1, parece obvio que lo que tenemos que hacer para acabar satisfaciendo la postcondición es multiplicar resul por n para obtener el multiplicatorio hasta n.

4. $resul = \prod_{i=1}^{n-1} i \rightarrow resul * n = \prod_{i=1}^{n} i$

5. $/*\, resul * n = \prod_{i=1}^{n} i \,*/\ \texttt{resul=resul*n;}\ /*\, resul = \prod_{i=1}^{n} i \,*/ (\textbf{\textit{AA}})$

6. $/*\, resul = \prod_{i=1}^{n-1} i \,*/\ \texttt{resul=resul*n;}\ /*\, resul = \prod_{i=1}^{n} i \,*/ (\textbf{\textit{RCN}}\ \textbf{4,5})$

7. $/*\, \Phi \wedge \neg B \,*/$
   `resul=ProductorioN(n-1);`
   `resul=resul*n;`
   $/*\, resul = \prod_{i=1}^{n} i \,*/ (\textbf{\textit{RCP}}\ \textbf{3,6})$

Ya hemos conseguido llegar a la postcondición.

iii) **Validación de la inducción:**

En este paso validaremos la inducción igual que lo hacíamos verificando programas recursivos.

$$E \equiv n$$

a) Caso simple: $\Phi \wedge B \equiv n \geq 0 \wedge n \leq 1 \rightarrow 0 \leq n \leq 1 \rightarrow E > -1$

b) Caso inductivo: $E' \equiv n - 1$

$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n > 1 \rightarrow E > 1 \wedge E' > 0$$

$$\Phi \wedge \neg B \equiv n \geq 0 \wedge n > 1 \rightarrow n > n - 1 \rightarrow E > E'$$

iv) **Programa documentado:**

Finalmente, escribimos la función completa con su especificación y el código derivado:

$ProductorioN(\text{int } n)\ return \text{ int } resul$

$PRE \equiv /* n \geq 0 */$

```
{
 if (n<=1)
 resul=1;
 else
 {
 resul=ProductorioN(n-1);
 resul=resul*n;
 }
}
```

$POST \equiv /* resul = \prod_{i=1}^{n} i */$

4.2 Derivación por inmersión

Dentro de la derivación de programas recursivos, estudiaremos un caso especial: **los derivados por inmersión**.

Diremos que un programa recursivo se ha derivado por inmersión cuando de la especificación propia del problema no sea posible derivar un programa recursivo y se haga necesario añadir nuevos parámetros para poder hacerlo recursivamente.

Un caso típico (no siendo el único) es el uso de vectores. Por ejemplo, para sumar todos los elementos de un vector la cabecera típica de la función recursiva sería:

$$SumaVector(\text{int}[\,]\,A)\,return\,\text{int sum}$$

Pero no podemos hacer esta función recursivamente tal como está, ya que no podemos descomponer el problema en trozos más sencillos hasta llegar a un caso básico. Entonces, ¿cómo podemos decirle a la llamada recursiva qué trozo del vector le corresponde sumar?.

La respuesta es: añadiendo nuevos parámetros. A esto se le llama derivar el programa por inmersión. Si, por ejemplo, le añadimos dos parámetros *i* y *j* tales que indiquen que sección del vector tiene que sumar cada llamada, obtendremos la **función inmersora**:

$$gSumaVector(\text{int}[\,]\,A,\text{int}\,i\,,\text{int}\,\,j)\,return\,\text{int gsum}$$

De esta manera, podremos resolver la suma de todo el vector pasando como *i* un 1 y como *j n:*

$$sumaVector(A)\equiv gSumaVector(A,1\,,n)$$

De cara a la verificación, como veremos a continuación sólo cambiará el primer paso, donde habrá que indicar la especificación original, la de la función inmersora y la llamada a la inmersora equivalente a la original.

Ejemplo: Derivar formalmente una función que sume todos los elementos de un vector.

i) **Especificación:**

a) La especificación original de la función:

$SumaVector(\text{int}[\,]A)\,return\,\text{int sum}$
$PRE \equiv /*\, n>0\,*/$
$POST \equiv /*\,\text{sum}=\sum_{k=1}^{n} A[k]\,*/$

b) La especificación de la función inmersora:

$gSumaVector(\text{int}[\,]A, \text{int}\, i, \text{int}\, j)\,return\,\text{int gsum}$
$PRE \equiv /*\, 1\le i\le j\le n\,*/$
$POST \equiv /*\, gsum=\sum_{k=i}^{j} A[k]\,*/$

c) La llamada a la función inmersora equivalente a la función original:

$sumaVector(A)\equiv gSumaVector(A,1,n)$

ii) **Análisis de casos:**

a) Caso simple:

Cogeremos i=j como caso simple, es decir, cuando la sección del vector a sumar tenga un único elemento, ese elemento será la suma de la sección.

1. $i=j\rightarrow\sum_{k=i}^{j} A[k]=A[i]$
2. $/*\sum_{k=i}^{j} A[k]=A[i]\,*/\ \mathbf{gsum=A[i];}\ /*\, gsum=\sum_{k=i}^{j} A[k]\,*/\,(\mathbf{AA})$
3. $/*\Phi\wedge B\,*/\ \mathbf{gsum=A[i];}\ /*\,\Psi\,*/\,(\mathbf{RCN\ 1,2})$

b) Caso inductivo:

Nuestra Hipótesis de Inducción será la siguiente:

$$/*\,1\le i+1\le j\le n\,*/\ gsum=gSumaVector(A,i+1,j);/*\,gsum=\sum_{k=i+1}^{j} A[k]\,*/$$

Sabiendo que en el caso inductivo se cumplirá siempre $\Phi\wedge\neg B$, siendo la condición del caso simple i=j, deduciremos el código del caso inductivo:

4. $1\le i\le j\le n\wedge i\neq j\rightarrow 1\le i+1\le j\le n$

$/*1 \le i+1 \le j \le n*/\ gsum = gSumaVector(A, i+1, j);$

5. $/*\ gsum = \sum_{k=i+1}^{j} A[k] \,*/ (\textbf{\textit{Hip.Ind.}})$

$/*1 \le i \le j \le n \wedge i \neq j\,*/\ gsum = gSumaVector(A, i+1, j);$

6. $/*\ gsum = \sum_{k=i+1}^{j} A[k] \,*/ (\textbf{\textit{RCN}}\ \textbf{4,5})$

Después de conseguir en gsum el sumatorio del vector desde *i+1* hasta *j*, sólo nos falta sumarle *A[i]* para poder calcular el sumatorio desde *i* hasta *j* (la postcondición):

7. $gsum = \sum_{k=i+1}^{j} A[k] \rightarrow gsum + A[i] = \sum_{k=i}^{j} A[k]$

8. $/*\ gsum + A[i] = \sum_{k=i}^{j} A[k] \,*/\ gsum = gsum + A[i]; /*\ gsum = \sum_{k=i}^{j} A[k] \,*/ (\textbf{\textit{AA}})$

9. $/*\ gsum = \sum_{k=i+1}^{j} A[k] \,*/\ gsum = gsum + A[i]; /*\ gsum = \sum_{k=i}^{j} A[k] \,*/ (\textbf{\textit{RCN}}\ \textbf{7,8})$

$/*1 \le i \le j \le n \wedge i \neq j\,*/$
`gsum = gSumaVector(A, i+1, j);`

10. `gsum = gsum + A[i];`

$/*\ gsum = \sum_{k=i}^{j} A[k] \,*/ (\textbf{\textit{RCP}}\ \textbf{6,9})$

iii) **Validación de la inducción:**

En las llamadas recursivas se incrementa en 1 la *i*, de manera que *i* se acerque cada vez más a *j*, hasta que finalmente (caso simple) sean iguales. Por ello, $E \equiv j - i$.

a) Caso simple: $\Phi \wedge B \equiv 1 \le i \le j \le n \wedge i = j \rightarrow j - i = 0 \rightarrow E > -1$
b) Caso inductivo: $E' \equiv j - (i+1)$
$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i \ge 1 \wedge j - (i+1) \ge 0 \rightarrow E > 0 \wedge E' > -1$
$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i > j - (i+1) \rightarrow E > E'$

iv) **Programa documentado:**

$gSumaVector(\text{int}[]A, \text{int}\ i, \text{int}\ j)\ return\ \text{int}\ gsum$
$PRE \equiv /*\ 1 \le i \le j \le n\ */$

```
{
if(i==j)
 gsum=A[i];
else
{
 gsum=gSumaVector(A,i+1,j);
 gsum=gsum+A[i];
}
```

$POST \equiv /*\ gsum = \sum_{k=i}^{j} A[k]\ */$

Ejercicios Derivación

Decidir en cada caso si hace falta utilizar inmersión o no:

1. Diseña un programa recursivo que calcule la **división entera x/y**.
2. Diseña un programa recursivo que **multiplique los cuadrados de todos los elementos de un vector** de números enteros.
3. Diseña un programa recursivo que calcule **la media de los elementos de un vector de números reales**.
4. Diseña un programa recursivo que calcule x^y .

Soluciones propuestas Derivación

1. Diseña un programa recursivo que calcule la **división entera x/y**.

1) Especificación:

$División(\text{int } x, \text{int } y)\ return\ \text{int div}$
$PRE \equiv /* x \geq 0 \wedge y > 0 */$
$POST \equiv /* \text{div} = x / y */$

2) Análisis de casos:

Caso simple: $x < y$

1. $\Phi \wedge B \equiv x \geq 0 \wedge y > 0 \wedge x < y \rightarrow x / y = \mathbf{0}$
2. $/* x / y = 0 */ \text{div} = 0\ /* x / y = \text{div} */ (\textbf{\textit{AA}})$
3. $/* \Phi \wedge B */ \text{div} = 0\ /* \Psi */ (\textbf{\textit{RCN}}\ \mathbf{1,2})$

Caso inductivo:

Hip.Ind.: $/* x - y \geq 0 \wedge y > 0 */\ \textbf{div} = \textbf{\textit{División}}(x - y, y);/* \textbf{div} = (x - y) / y */$

4. $\Phi \wedge \neg B \equiv x \geq 0 \wedge y > 0 \wedge x \geq y \rightarrow x - y \geq 0$
5. $/* x - y \geq 0 \wedge y > 0 */ \texttt{div} = \texttt{División}(x - y, y);$
   $/* \text{div} = (x - y) / y */ (\textbf{\textit{Hip.Ind.}})$
6. $/* \Phi \wedge \neg B */ \texttt{div} = \texttt{División}(x - y, y);$
   $/* \text{div} = (x - y) / y */ (\textbf{\textit{RCN}}\ \mathbf{4,5})$
7. $\text{div} = (x - y) / y \rightarrow \text{div} + 1 = x / y$
8. $/* \text{div} + 1 = x / y */ \texttt{div} = \texttt{div} + 1;$
   $/* \text{div} = x / y */ (\textbf{\textit{AA}})$
9. $/* \text{div} = (x - y) / y */ \texttt{div} = \texttt{div} + 1;$
   $/* \text{div} = x / y */ (\textbf{\textit{RCN}}\ \mathbf{7,8})$
10. $/* \Phi \wedge \neg B */ P_i;$
    $/* \Psi */ (\textbf{\textit{RCP}}\ \mathbf{6,9})$

3) Validación de la inducción:

En este caso, la expresión cota más sencilla será *x*, que irá decreciendo entre las diferentes llamadas recursivas y siempre será un número natural. Por lo tanto, **E=x**

a) Caso simple: $\Phi \wedge B \equiv x \geq 0 \wedge y > 0 \wedge x < y \rightarrow x \geq 0 \rightarrow E > -1$
b) Caso inductivo: $E' \equiv x - y$
$\Phi \wedge \neg B \equiv x \geq 0 \wedge y > 0 \wedge x \geq y \rightarrow x \geq 0 \wedge x - y \geq 0 \rightarrow E > -1 \wedge E' > -1$
$\Phi \wedge \neg B \equiv x \geq 0 \wedge y > 0 \wedge x \geq y \rightarrow x > x - y \rightarrow E > E'$

4) Programa documentado:

División(int x, int y) *return* int div
PRE ≡ /* $x \geq 0 \wedge y > 0$ */
{
```
if(x<y)
 div=0;
else
{
 div=División(x-y,y);
 div=div+1;
}
```
POST ≡ /* div = x / y */

2. Diseña un programa recursivo que **multiplique los cuadrados de todos los elementos de un vector** de números enteros.

1) Especificación:

1. $MultiCuad(\text{int }[]\,A)\,return\,\text{int resul}$
   $PRE \equiv /* n \geq 1 */$
   $POST \equiv /* resul = \prod_{k=1}^{n} A[k]^2 */$

2. $gMultiCuad(\text{int }[]\,A, \text{int } i, \text{int } j)\,return\,\text{int gresul}$
   $PRE \equiv /* 1 \leq i \leq j \leq n */$
   $POST \equiv /* gresul = \prod_{k=i}^{j} A[k]^2 */$

3. $MultiCuad(A) \equiv gMultiCuad(A, 1, n)$

2) Análisis de casos:

Caso simple: $i = j$

1. $\Phi \wedge B \equiv 1 \leq i \leq j \leq n \wedge i = j \rightarrow 1 \leq i = j \leq n \rightarrow \prod_{k=i}^{j} A[k]^2 = \boldsymbol{A[i]^2}$

2. $/* \prod_{k=i}^{j} A[k]^2 = \boldsymbol{A[i]} * \boldsymbol{A[i]} */\ gresul = A[i] * A[i]; /* \prod_{k=i}^{j} A[k]^2 = gresul */$

3. $/* \Phi \wedge B */\ gresul = A[i] * A[i]; /* \Psi */ (\textbf{\textit{RCN}}\ \mathbf{1,2})$

Caso inductivo:

Hip.Ind.: $/* \mathbf{1 \leq i+1 \leq j \leq n} */$ `gresul=gMultiCuad(A,i+1,j);`

$$/* \textbf{\textit{gresul}} = \prod_{\boldsymbol{k=i+1}}^{\boldsymbol{j}} \boldsymbol{A[k]^2} */$$

4. $\Phi \wedge \neg B \equiv 1 \leq i \leq j \leq n \wedge i \neq j \rightarrow 1 \leq i+1 \leq j \leq n$
   $/* 1 \leq i+1 \leq j \leq n */$ `gresul=gMultiCuad(A,i+1,j);`

5. $/* gresul = \prod_{k=i+1}^{j} A[k]^2 */ (\textbf{\textit{Hip.Ind.}})$
   $/* \Phi \wedge \neg B */$ `gresul=gMultiCuad(A,i+1,j);`

6. $/* gresul = \prod_{k=i+1}^{j} A[k]^2 */ (\textbf{\textit{RCN}}\ \mathbf{4,5})$

7. $gresul = \prod_{k=i+1}^{j} A[k]^2 \rightarrow gresul * A[i]^2 = \prod_{k=i}^{j} A[k]^2$

8. $/* gresul * A[i]^2 = \prod_{k=i}^{j} A[k]^2 */$ `gresul=gresul*A[i]*A[i];`
   $/* gresul = \prod_{k=i}^{j} A[k]^2 */ (\boldsymbol{AA})$
9. $/* gresul = \prod_{k=i+1}^{j} A[k]^2 */$ `gresul=gresul*A[i]*A[i];`
   $/* \Psi */ (\boldsymbol{RCN}\ \mathbf{7,8})$
10. $/* \Phi \wedge \neg B */$ `P`$_i$`;`
    $/* \Psi */ (\boldsymbol{RCP}\ \mathbf{6,9})$

3) Validación de la inducción:

La expresión cota que utilizaremos será *j-i*, ya que la *i* se acercará a *j* hasta que sean iguales. Por lo tanto, $E \equiv j - i$

a) Caso simple: $\Phi \wedge B \equiv 1 \le i \le j \le n \wedge i = j \rightarrow j - i = 0 \rightarrow E > -1$
b) Caso inductivo: $E' \equiv j - (i+1)$

$$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i > 0 \wedge j - (i+1) \ge 0 \rightarrow E > 0 \wedge E' > -1$$
$$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i > j - (i+1) \rightarrow E > E'$$

4) Programa documentado:

$gMultiCuad(\text{int}[]\ A, \text{int}\ i, \text{int}\ j)\ return\ \text{int}\ gresul$
$PRE \equiv /* 1 \le i \le j \le n */$

```
{
if(i==j)
 gsum=A[i]*A[i];
else
{
 gsum=gMultiCuad(A,i+1,j);
 gsum=gsum*A[i]*A[i];
}
```

$POST \equiv /* gsum = \prod_{k=i}^{j} A[k]^2 */$

3. Diseña un programa recursivo que calcule **la media de los elementos de un vector**.

1) Especificación:

1. $Media(\text{float } [] A) \; return \; \text{float resul}$
   $PRE \equiv /* n \geq 1 */$
   $POST \equiv /* resul = \sum_{k=1}^{n} (A[k]/n) */$

2. $gMedia(\text{float } [] A, \text{int } i, \text{int } j) \; return \; \text{float gresul}$
   $PRE \equiv /* 1 \leq i \leq j \leq n */$
   $POST \equiv /* gresul = \sum_{k=i}^{j} (A[k]/n) */$

3. $Media(A) \equiv gMedia(A, 1, n)$

2) Análisis de casos:

Caso simple: $i = j$

1. $\Phi \wedge B \equiv 1 \leq i \leq j \leq n \wedge i = j \rightarrow 1 \leq i = j \leq n \rightarrow \sum_{k=i}^{j} (A[k]/n) = \mathbf{A[i]/n}$

2. $/* \sum_{k=i}^{j} (A[k]/n) = \mathbf{A[i]/n} */ \; \mathbf{gresul = A[i]/n;} \; /* \sum_{k=i}^{j} (A[k]/n) = gresul */ (\mathbf{AA})$

3. $/* \Phi \wedge B */ \; \mathbf{gresul = A[i]/n;} \; /* \Psi */ (\mathbf{RCN \; 1,2})$

Caso inductivo:

Hip.Ind.: $/* \mathbf{1 \leq i+1 \leq j \leq n} */ \; \mathbf{gresul = gMedia(A, i+1, j);}$

$$/* \mathbf{gresul} = \sum_{\mathbf{k=i+1}}^{\mathbf{j}} (\mathbf{A[k]/n}) */$$

4. $\Phi \wedge \neg B \equiv 1 \leq i \leq j \leq n \wedge i \neq j \rightarrow 1 \leq i+1 \leq j \leq n$
   $/* 1 \leq i+1 \leq j \leq n */ \; gresul = gMedia(A, i+1, j);$

5. $/* gresul = \sum_{k=i+1}^{j} (A[k]/n) */ (\mathbf{Hip.Ind.})$
   $/* \Phi \wedge \neg B */ \; gresul = gMedia(A, i+1, j);$

6. $/* gresul = \sum_{k=i+1}^{j} (A[k]/n) */ (\mathbf{RCN \; 4,5})$

7. $gresul = \sum_{k=i+1}^{j} (A[k]/n) \rightarrow gresul + (A[i]/n) = \sum_{k=i}^{j} (A[k]/n)$

8. /* $gresul + (A[i]/n) = \sum_{k=i}^{j} (A[k]/n)$ */ `gresul=gresul+(A[i]/n);`

   /* $gresul = \sum_{k=i}^{j} (A[k]/n)$ */ (***AA***)

9. /* $gresul = \sum_{k=i+1}^{j} (A[k]/n)$ */ `gresul=gresul+(A[i]/n);`

   /* $gresul = \sum_{k=i}^{j} (A[k]/n)$ */ (***RCN* 7,8**)

10. /* $\Phi \wedge \neg B$ */ P_i; /* Ψ */ (***RCP* 6,9**)

3) Validación de la inducción:

La expresión cota que utilizaremos será *j*-*i*, ya que la *i* se acercará a *j* hasta que sean iguales. Por lo tanto, $E \equiv j - i$

a) Caso simple: $\Phi \wedge B \equiv 1 \le i \le j \le n \wedge i = j \rightarrow j - i = 0 \rightarrow E > -1$

b) Caso inductivo: $E' \equiv j - (i+1)$

$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i > 0 \wedge j - (i+1) \ge 0 \rightarrow E > 0 \wedge E' > -1$

$\Phi \wedge \neg B \equiv 1 \le i \le j \le n \wedge i \neq j \rightarrow j - i > j - (i+1) \rightarrow E > E'$

4) Programa documentado:

$gMedia(\text{float}[\,]A, \text{int}\ i, \text{int}\ j)\ return\ \text{float}\ gresul$

$PRE \equiv /* 1 \le i \le j \le n */$

```
{
if(i==j)
 gresul=A[i]/n;
else
{
 gresul=gSumaVector(A,i+1,j);
 gresul=gresul+(A[i]/n);
}
```

$POST \equiv /* gresul = \sum_{k=i}^{j} (A[k])/n */$

4. Diseña un programa recursivo que calcule x^y .

1) Especificación:

$Potencia(\text{int } x, \text{int } y)\ return\ \text{int pot}$
$PRE \equiv /*\ y \geq 0\ */$
$POST \equiv /*\ pot = x^y\ */$

2) Análisis de casos:

Caso simple: $y = 0$

1. $\Phi \wedge B \equiv y \geq 0 \wedge y = 0 \rightarrow y = 0 \rightarrow x^y = \mathbf{1}$
2. $/*\ x^y = 1\ */\ \mathtt{pot = 1;}\ /*\ pot = x^y\ */ (\mathbf{AA})$
3. $/*\ \Phi \wedge B\ */\ \mathtt{pot = 1;}\ /*\ \Psi\ */ (\mathbf{RCN\ 1,2})$

Caso inductivo:

Hip.Ind.: $/*\ y-1 \geq 0\ */\ pot = Potencia(x, y-1);\ /*\ pot = x^{y-1}\ */$

4. $\Phi \wedge \neg B \equiv y \geq 0 \wedge y \neq 0 \rightarrow y > 0 \rightarrow y - 1 \geq 0$
5. $/*\ y-1 \geq 0\ */\ pot = Potencia(x, y-1);\ /*\ pot = x^{y-1}\ */ (\mathbf{Hip.Ind.})$
6. $/*\ \Phi \wedge \neg B\ */\ pot = Potencia(x, y-1);\ /*\ pot = x^{y-1}\ */ (\mathbf{RCN\ 4,5})$
7. $pot = x^{y-1} \rightarrow pot * x = x^y$
8. $/*\ pot * x = x^y\ */\ \mathtt{pot = pot * x;}$
   $/*\ pot = x^y\ */ (\mathbf{AA})$
9. $/*\ pot = x^{y-1}\ */\ \mathtt{pot = pot * x;}$
   $/*\ pot = x^y\ */ (\mathbf{RCN\ 7,8})$
10. $/*\ \Phi \wedge \neg B\ */\ \mathtt{P_i;}$
    $/*\ \Psi\ */ (\mathbf{RCP\ 6,9})$

3) Validación de la inducción:

En este caso, la expresión cota más sencilla será *y*, que irá decreciendo entre las diferentes llamadas recursivas hasta llegar a *0*. Por lo tanto, **E=y**

a) Caso simple: $\Phi \wedge B \equiv y \geq 0 \wedge y = 0 \rightarrow y = 0 \rightarrow E > -1$
b) Caso inductivo: $E' \equiv y - 1$

$$\Phi \wedge \neg B \equiv y \geq 0 \wedge y \neq 0 \rightarrow y > 0 \rightarrow E > 0 \wedge E' > -1$$
$$\Phi \wedge \neg B \equiv y \geq 0 \wedge y \neq 0 \rightarrow y > 0 \rightarrow y > y - 1 \rightarrow E > E'$$

4) Programa documentado:

$Potencia(\text{int } x, \text{int } y)\ return \text{int } pot$
$PRE \equiv /* \ y \geq 0 \ */$
{
`if(y==0)`
`pot=1;`
`else`
{
`pot=Potencia(x,y-1);`
`pot=pot*x;`
}
$POST \equiv /* \ pot = x^y \ */$